UFES - Universidade Federal do Espírito Santo

Engenharia de Software

Notas de Aula

PARTE II

Ricardo de Almeida Falbo

Monalessa Perini Barcellos

Curso: **Engenharia da Computação**

Adaptação, em 2011, por Monalessa Perini Barcellos das Notas de Aula de Engenharia de Requisitos e Projeto de Sistemas de Ricardo de Almeida Falbo

Observação: Este material aborda conteúdos que vão além do programa da disciplina Engenharia de Software para o curso Engenharia da Computação. No entanto, optou-se por incluir esses conteúdos para que alunos que tenham interesse em se aprofundar no tema possam usar este material como uma referência.

ÍNDICE

Organização do Texto

Este material é parte integrante das Notas de Aula da disciplina Engenharia de Software, as quais encontram-se divididas em duas partes.

Na **Parte I** foram abordados o processo de software em si e as atividades de gerência de projetos e de garantia da qualidade. Os capítulos da Parte I são:

- Capítulo 1 – *Introdução* – capítulo de introdução ao conteúdo.
- Capítulo 2 – *Processo de Software* – enfoca os processos de software, os elementos que compõem um processo, a definição de processos para projetos, modelos de processo, normas e modelos de qualidade de processo de software e a automatização do processo de software.
- Capítulo 3 –*Gerência de Projetos* – são abordadas as principais atividades da gerência de projetos, a saber: definição do escopo do projeto, estimativas, análise de riscos, elaboração de cronograma, elaboração do plano de projeto e acompanhamento de projetos.
- Capítulo 4 – *Gerência da Qualidade* – trata de algumas atividades relacionadas à garantia da qualidade, incluindo a medição e métricas associadas, revisões e inspeções e a gerência de configuração de software.

Este material constitui a **Parte II** das Notas de Aula de Engenharia de Software e aborda as atividades de desenvolvimento. Contém quatro capítulos:

- Capítulo 5 – *Especificação e Análise de Requisitos* – são discutidos o que é um requisito de software e tipos de requisitos. Em seguida, são abordadas a especificação e a análise de requisitos. É oferecida uma visão geral da abordagem de Análise Orientada a Objetos. As técnicas de modelagem de casos de uso, modelagem estrutural e modelagem de estados são apresentadas.
- Capítulo 6 – *Projeto de Sistema* – aborda os conceitos básicos de projeto de sistemas, tratando da arquitetura do sistema a ser desenvolvido e do projeto de seus componentes. Também são discutidos o projeto de dados e o projeto de interface com o usuário.
- Capítulo 7 – *Implementação e Testes* – são enfocadas as atividades de implementação e testes, sendo esta última tratada em diferentes níveis, a saber: teste de unidade, teste de integração, teste de validação e teste de sistema.
- Capítulo 8 – *Entrega e Manutenção* – discute as questões relacionadas à entrega do sistema para o cliente, tais como o treinamento e a documentação de entrega, e a atividade de manutenção do sistema.

Capítulo 5 – Especificação e Análise de Requisitos

Uma vez estudadas as atividades de gerência de projetos (Capítulo 3) e de garantia de qualidade (Capítulo 4), podemos passar a discutir as atividades do processo de desenvolvimento de software, ou seja, aquelas atividades que contribuem diretamente para o desenvolvimento do produto de software a ser entregue ao cliente e que, portanto, formam a espinha dorsal do desenvolvimento.

No que tange às atividades técnicas do desenvolvimento de software, a primeira coisa a ser feita é capturar os requisitos que o sistema a ser desenvolvido tem de tratar. Um completo entendimento dos requisitos do software é essencial para o sucesso de um esforço de desenvolvimento de software. A Engenharia de Requisitos é um processo de descoberta, refinamento, modelagem e especificação. O escopo do software definido no planejamento do projeto é refinado em detalhe, as funções e o desempenho do software são especificados, as interfaces com outros sistemas são indicadas e restrições que o software deve atender são estabelecidas. Modelos dos dados requeridos, do controle e do comportamento operacional são construídos. Finalmente, critérios para a avaliação da qualidade em atividades subsequentes são estabelecidos.

Este capítulo é bastante extenso e encontra-se assim organizado: na seção 5.1 são apresentados o conceito de requisitos e o processo de Engenharia de Requisitos; na seção 5.2 o paradigma orientado a objetos, à luz do qual as atividades do processo de desenvolvimento são discutidas neste texto, é apresentado; na seção 5.3 são discutidos o levantamento e registro dos requisitos; a seção 5.4 trata da modelagem de caso de uso; a seção 5.5 trata da modelagem conceitual estrutura; e, por fim, a seção 5.6 aborda os diagramas de estados.

5.1 – Engenharia de Requisitos de Software

Uma das principais medidas do sucesso de um software é o grau no qual ele atende aos objetivos e requisitos para os quais foi construído. De forma geral, a Engenharia de Requisitos de Software é o processo de identificar todos os envolvidos, descobrir seus objetivos e necessidades e documentá-los de forma apropriada para análise, comunicação e posterior implementação (TOGNERI, 2002). Mas antes de voltar a atenção para as atividades do processo de Engenharia de Requisitos, é importante entender o que é um requisito.

5.1.1 - Requisito e Tipos de Requisitos

As descrições das funções que um sistema deve incorporar e das restrições que devem ser satisfeitas são os requisitos para o sistema. Isto é, os requisitos de um sistema definem o que o sistema deve fazer e as circunstâncias sob as quais deve operar. Em outras palavras, os requisitos definem os serviços que o sistema deve fornecer e dispõem sobre as restrições à operação do mesmo (SOMMERVILLE, 2007).

Requisitos são, normalmente, classificados em **requisitos funcionais** e **requisitos não funcionais**. Requisitos funcionais, como o próprio nome indica, apontam as funções que o sistema deve fornecer e como o sistema deve se comportar em determinadas situações. Já os requisitos não funcionais descrevem restrições sobre

as funções oferecidas, tais como restrições de tempo, de uso de recursos etc. Alguns requisitos não funcionais dizem respeito ao sistema como um todo e não a funcionalidade específica. Dependendo da natureza, os requisitos não funcionais podem ser classificados de diferentes maneiras, tais como requisitos de desempenho, requisitos de portabilidade, requisitos legais, requisitos de conformidade etc (SOMMERVILLE, 2007).

5.1.2 - O Processo da Engenharia de Requisitos

O processo de descobrir, analisar, documentar e verificar / validar requisitos é chamado de processo de engenharia de requisitos (SOMMERVILLE, 2007). Os processos de engenharia de requisitos variam muito de uma organização para outra, mas, de maneira geral, a maioria dos processos de Engenharia de Requisitos é composta das seguintes atividades (TOGNERI, 2002):

- Levantamento (ou Descoberta ou Elicitação) de Requisitos: Nesta fase, os usuários, clientes e especialistas de domínio são identificados e trabalham junto com os engenheiros de requisitos para descobrir, articular e entender a organização como um todo, o domínio da aplicação, os processos de negócio específicos, as necessidades que o software deve atender e os problemas e deficiências dos sistemas atuais. Os diferentes pontos de vista dos participantes do processo, bem como as oportunidades de melhoria e restrições existentes, os problemas a serem resolvidos, o desempenho requerido e as restrições também devem ser levantados.
- Análise de Requisitos: visa a estabelecer um conjunto acordado de requisitos consistentes e sem ambiguidades, que possa ser usado como base para o desenvolvimento do software. Para tal, diversos tipos de modelos são construídos. Geralmente, a análise de requisitos inclui também a negociação para resolver conflitos detectados.
- Documentação de Requisitos: é a atividade de representar os resultados da Engenharia de Requisitos em um documento (ou conjunto de documentos), contendo os requisitos do software.
- Verificação e Validação de Requisitos: A verificação de requisitos avalia se o documento de Especificação de Requisitos está sendo construído de forma correta, de acordo com padrões previamente definidos, sem conter requisitos ambíguos, incompletos ou, ainda, requisitos incoerentes ou impossíveis de serem testados. Já a validação diz respeito a avaliar se esse documento está correto, ou seja, se os requisitos e modelos documentados atendem às reais necessidades e requisitos dos usuários / cliente.
- Gerência de Requisitos: se preocupa em gerenciar as mudanças nos requisitos já acordados, manter uma trilha dessas mudanças, gerenciar os relacionamentos entre os requisitos e as dependências entre o documento de requisitos e os demais artefatos produzidos no processo de software, de forma a garantir que mudanças nos requisitos sejam feitas de maneira controlada e documentada.

É possível notar que, das cinco atividades do processo de Engenharia de Requisitos anteriormente listadas, as três últimas são, na realidade, instanciações para a Engenharia de Requisitos de atividades genéricas já discutidas no capítulo 4, a saber:

documentação, garantia da qualidade e gerência de configuração. Assim sendo, neste capítulo, somente as duas primeiras atividades (Levantamento e Análise de Requisitos) são discutidas.

O **levantamento de requisitos** é uma atividade complexa que não se resume somente a perguntar às pessoas o que elas desejam e também não é uma simples transferência de conhecimento. Várias técnicas de levantamento de requisitos são normalmente empregadas pelos engenheiros de requisitos (ou analistas de sistemas), dentre elas entrevistas, questionários, prototipação, investigação de documentos, observação, dinâmicas de grupo etc.

Uma característica fundamental da atividade de levantamento de requisitos é o seu enfoque em uma visão do cliente / usuário. Em outras palavras, está-se olhando para o sistema a ser desenvolvido por uma perspectiva externa. Ainda não se está procurando definir a estrutura interna do sistema, mas sim procurando saber que funcionalidades o sistema deve oferecer ao usuário e que restrições elas devem satisfazer.

A **análise de requisitos** (muitas vezes chamada análise de sistemas), por outro lado, enfoca a estrutura interna do sistema. Ou seja, procura-se definir o que o sistema tem de ter internamente para tratar adequadamente os requisitos levantados. Assim sendo, a análise de requisitos é, em última instância, uma atividade de construção de modelos. Um modelo é uma representação de alguma coisa do mundo real, uma abstração da realidade, e, portanto, representa uma seleção de características do mundo real relevantes para o propósito do sistema em questão.

Modelos são fundamentais no desenvolvimento de sistemas. Tipicamente eles são construídos para:

- possibilitar o estudo do comportamento do sistema;
- facilitar a comunicação entre os componentes da equipe de desenvolvimento e clientes e usuários;
- possibilitar a discussão de correções e modificações com o usuário;
- formar a documentação do sistema.

Um modelo enfatiza um conjunto de características da realidade, que corresponde à *dimensão do modelo*. Além da dimensão que um modelo enfatiza, modelos possuem níveis de abstração. O *nível de abstração* de um modelo diz respeito ao grau de detalhamento com que as características do sistema são representadas. Em cada nível há uma ênfase seletiva nos detalhes representados. No caso do desenvolvimento de sistemas, geralmente, são considerados três níveis:

- *conceitual*: considera características do sistema independentes do ambiente computacional (hardware e software) no qual o sistema será implementado. Essas características são dependentes unicamente das necessidades do usuário. Modelos conceituais são construídos na atividade de análise de requisitos.
- *lógico*: características dependentes de um determinado *tipo* de sistema computacional. Essas características são, contudo, independentes de produtos específicos. Tais modelos são típicos da fase de projeto.
- *físico*: características dependentes de um sistema computacional específico, isto é, uma linguagem e um compilador específico, um sistema gerenciador de bancos de dados específico, o hardware de um determinado fabricante etc. Tais

modelos podem ser construídos tanto na fase de projeto quanto na fase de implementação.

Conforme apontado acima, nas primeiras etapas do processo de desenvolvimento (levantamento e análise de requisitos), o engenheiro de software representa o sistema através de modelos conceituais. Nas etapas posteriores, as características lógicas e físicas são representadas em novos modelos.

Para conduzir o processo de engenharia de requisitos, sobretudo a modelagem conceitual, é necessário adotar um paradigma de desenvolvimento. Paradigmas de desenvolvimento estabelecem a forma de se ver o mundo e, portanto, definem as características básicas dos modelos a serem construídos. Por exemplo, o paradigma orientado a objetos parte do pressuposto que o mundo é povoado por objetos, ou seja, a abstração básica para se representar as coisas do mundo são os objetos. Já o paradigma estruturado adota uma visão de desenvolvimento baseada em um modelo entrada-processamento-saída. No paradigma estruturado, os dados são considerados separadamente das funções que os transformam e a decomposição funcional é usada intensamente.

Neste texto, discutimos as atividades do processo de desenvolvimento de software à luz do paradigma orientado a objetos. Assim sendo, as atividades de levantamento e análise de requisitos são conduzidas tomando por base esse paradigma. Na próxima seção são apresentados os principais conceitos do paradigma orientado a objetos.

5.2 – O Paradigma Orientado a Objetos

Um dos paradigmas de desenvolvimento mais adotados atualmente é a orientação a objetos. Segundo esse paradigma, o mundo é visto como sendo composto por *objetos*, onde um objeto é uma entidade que combina estrutura de dados e comportamento funcional. No paradigma orientado a objetos, os sistemas são modelados como um número de objetos que interagem.

A orientação a objetos oferece um número de conceitos bastante apropriados para a modelagem de sistemas. Os modelos baseados em objetos são úteis para a compreensão de problemas, para a comunicação com os especialistas e usuários das aplicações, e para a realização das tarefas ao longo do processo de desenvolvimento de software. Em um sistema construído segundo o paradigma orientado a objetos (OO), componentes são partes encapsuladas de dados e funções, que podem herdar atributos e comportamento de outros componentes da mesma natureza e cujos componentes comunicam-se entre si por meio de mensagens (YOURDON, 1994). O paradigma OO utiliza uma perspectiva humana de observação da realidade, incluindo objetos, classificação e compreensão hierárquica.

Para fazer bom uso do paradigma OO, é importante conhecer os princípios adotados por ele na administração da complexidade, bem como seus principais conceitos.

5.2.1 – Princípios para Administração da Complexidade

O mundo real é algo extremamente complexo. Quanto mais de perto o observamos, mais claramente percebemos sua complexidade. A orientação a objetos

tenta gerenciar a complexidade inerente dos problemas do mundo real, abstraindo conhecimento relevante e encapsulando-o dentro de objetos. De fato, alguns princípios básicos gerais para a administração da complexidade norteiam o paradigma orientado a objetos, entre eles abstração, encapsulamento e modularidade.

Abstração

Uma das principais formas do ser humano lidar com a complexidade é através do uso de abstrações. As pessoas tipicamente tentam compreender o mundo, construindo modelos mentais de partes dele. Tais modelos são uma visão simplificada de algo, onde apenas os elementos relevantes são considerados. Modelos, portanto, são mais simples do que os complexos sistemas que eles modelam.

Seja o exemplo de um mapa representando um modelo de um território. Um mapa é útil porque abstrai apenas as características do território que se deseja modelar. Se um mapa incluísse todos os detalhes do território, provavelmente teria o mesmo tamanho do território e, portanto, não serviria a seu propósito.

Da mesma forma que um mapa precisa ser significativamente menor que o território que mapeia, incluindo apenas as informações selecionadas, um modelo deve abstrair apenas as características relevantes de um sistema para seu entendimento. Assim, pode-se definir abstração como sendo o princípio de ignorar aspectos não relevantes de um assunto, segundo a perspectiva de um observador, tornando possível uma concentração maior nos aspectos principais do mesmo. De fato, a abstração consiste na seleção que um observador faz de alguns aspectos de um assunto, em detrimento de outros que não demonstram ser relevantes para o propósito em questão. A Figura 5.1 ilustra o conceito de abstração.
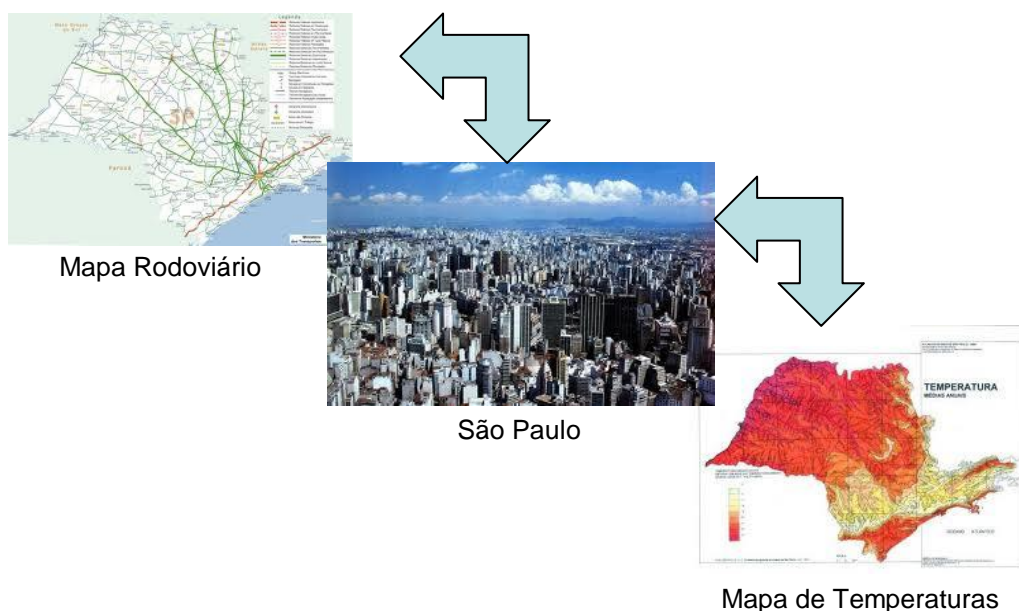
Figura 5.1 – Ilustração do Conceito de Abstração.

Um conjunto de abstrações pode formar uma hierarquia e, pela identificação dessas hierarquias, é possível simplificar significativamente o entendimento sobre um problema (BOOCH, 1994). Assim, hierarquia é uma forma interessante de arrumar as abstrações.

Encapsulamento

No mundo real, um objeto pode interagir com outro sem conhecer seu funcionamento interno. Uma pessoa, por exemplo, utiliza um forno de microondas sem saber efetivamente qual a sua estrutura interna ou como seus mecanismos internos são ativados. Para utilizá-lo, basta saber usar seu painel de controle (a interface do aparelho) para realizar as operações de ligar/desligar, informar o tempo de cozimento etc. Como essas operações produzem os resultados, não interessa ao cozinheiro. A Figura 5.2 ilustra o conceito de encapsulamento.

Estrutura Interna

Interface
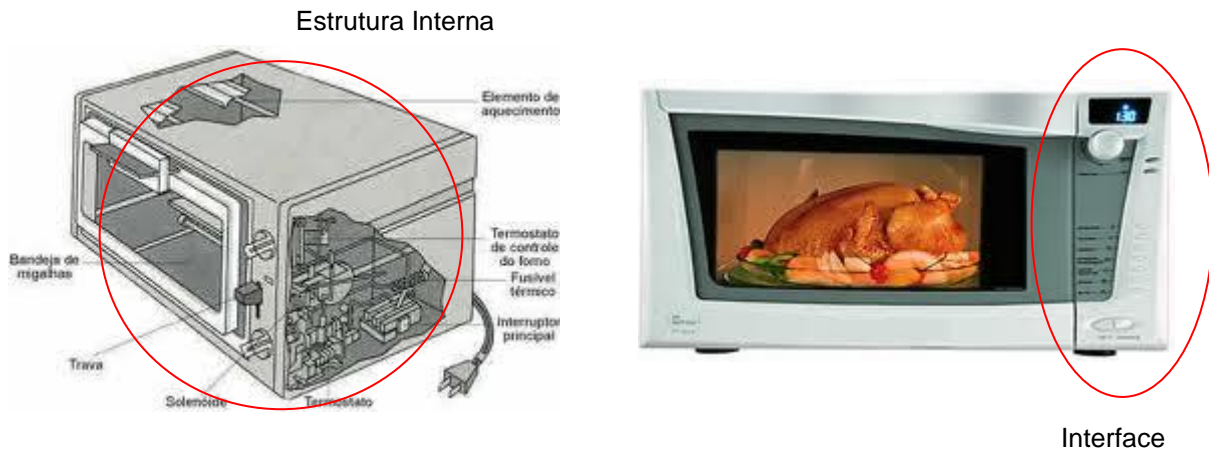
Figura 5.2 – Ilustração do Conceito de Encapsulamento.

O encapsulamento consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, de seus detalhes internos de implementação, que ficam ocultos dos demais objetos (BLAHA; RUMBAUGH, 2006). A interface de um objeto deve ser definida de forma a revelar o menos possível sobre o seu funcionamento interno.

Abstração e encapsulamento são conceitos complementares: enquanto a abstração enfoca o comportamento observável de um objeto, o encapsulamento oculta a implementação que origina esse comportamento. Encapsulamento é frequentemente conseguido através da ocultação de informação, isto é, escondendo detalhes que não contribuem para suas características essenciais. Tipicamente, em um sistema orientado a objetos, a estrutura de um objeto e a implementação de seus métodos são encapsuladas (BOOCH, 1994). Assim, o encapsulamento serve para separar a interface contratual de uma abstração e sua implementação. Os usuários têm conhecimento apenas das operações que podem ser requisitadas e precisam estar cientes apenas do *quê* as operações realizam e não *como* elas estão implementadas.

A principal motivação para o encapsulamento é facilitar a reutilização de objetos e garantir estabilidade aos sistemas. Um encapsulamento bem feito pode servir de base para a localização de decisões de projeto que necessitam ser alteradas. Uma operação pode ter sido implementada de maneira ineficiente e, portanto, pode ser necessário escolher um novo algoritmo. Se a operação está encapsulada, apenas o objeto que a define precisa ser modificado, garantindo estabilidade ao sistema.

Modularidade

Os métodos de desenvolvimento de software buscam obter sistemas modulares, isto é, construídos a partir de elementos que sejam autônomos e conectados por uma estrutura simples e coerente. Modularidade visa à obtenção de sistemas decompostos em um conjunto de módulos coesos e fracamente acoplados e é crucial para se obter reusabilidade e facilidade de extensão.

Abstração, encapsulamento e modularidade são princípios sinergéticos, isto é, ao se trabalhar bem com um deles, estão-se aperfeiçoando os outros também.

5.2.2 – Principais Conceitos da Orientação a Objetos

De maneira simples, o paradigma OO traz uma visão de mundo em que os fenômenos e domínios são vistos como coleções de objetos interagindo entre si. Essa forma simples de se colocar a visão do paradigma OO esconde conceitos importantes da orientação a objetos que são a base para o desenvolvimento OO, tais como classes, associações, generalização etc. A seguir os principais conceitos da orientação a objetos são discutidos.

Objetos

O mundo real é povoado por entidades que interagem entre si, onde cada uma delas desempenha um papel específico. Na orientação a objetos, essas entidades são ditas *objetos*. Objetos podem ser coisas concretas ou abstratas, tais como um carro, uma reserva de passagem aérea, uma organização etc.

Do ponto de vista da modelagem de sistemas, um objeto é uma entidade que incorpora uma abstração relevante no contexto de uma aplicação. Um objeto possui um estado (informação), exibe um comportamento bem definido (dado por um número de operações para examinar ou alterar seu estado) e tem identidade única.

O estado de um objeto compreende o conjunto de suas propriedades, associadas a seus valores correntes. Propriedades de objetos são geralmente referenciadas como atributos e associações. Portanto, o estado de um objeto diz respeito aos seus atributos/associações e aos valores a eles associados.

A abstração incorporada por um objeto é caracterizada por um conjunto de serviços ou operações que outros objetos, ditos clientes, podem requisitar. Operações são usadas para recuperar ou manipular a informação de estado de um objeto e se referem apenas às estruturas de dados do próprio objeto, não devendo acessar diretamente estruturas de outros objetos. Caso a informação necessária para a realização de uma operação não esteja disponível, o objeto terá de colaborar com outros objetos.

A comunicação entre objetos dá-se por meio de *troca de mensagens*. Para requisitar uma operação de um objeto, é necessário enviar uma mensagem para ele. Uma mensagem consiste do nome da operação sendo requisitada e os argumentos requeridos. Assim, o comportamento de um objeto representa como esse objeto reage às mensagens a ele enviadas. Em outras palavras, o conjunto de mensagens a que um objeto pode responder representa o seu comportamento. Um objeto é, pois, uma entidade que tem seu estado representado por um conjunto de atributos (uma estrutura de informação) e seu comportamento representado por um conjunto de operações.

Cada objeto tem uma identidade própria, que lhe é inerente. Todos os objetos têm existência própria, ou seja, dois objetos são distintos, mesmo se seus estados e comportamentos forem iguais. A identidade de um objeto transcende os valores correntes de suas propriedades.

Classes

No mundo real, diferentes objetos desempenham um mesmo papel. Seja o caso de duas cadeiras. Apesar de serem objetos diferentes, elas compartilham uma mesma estrutura e um mesmo comportamento. Entretanto, não há necessidade de se despender tempo modelando cada cadeira. Basta definir, em um único lugar, um modelo descrevendo a estrutura e o comportamento desses objetos. A esse modelo dá-se o nome de *classe*. Uma classe descreve um conjunto de objetos com as mesmas propriedades (atributos e associações), o mesmo comportamento (operações) e a mesma semântica. Objetos que se comportam da maneira especificada pela classe são ditos *instâncias* dessa classe.

Todo objeto pertence a uma classe, ou seja, é instância de uma classe. De fato, a orientação a objetos norteia o processo de desenvolvimento através da *classificação de objetos*, isto é, objetos são agrupados em classes, em função de exibirem facetas similares, sem, no entanto, perda de sua individualidade, como ilustra a Figura 5.3. Assim, do ponto de vista estrutural, a modelagem orientada a objetos consiste, basicamente, na definição de classes. O comportamento e a estrutura de informação de uma instância são definidos pela sua classe.

Objetos com propriedades e comportamento idênticos são descritos como instâncias de uma mesma classe, de modo que a descrição de suas propriedades possa ser feita uma única vez, de forma concisa, independentemente do número de objetos que tenham tais propriedades em comum. Deste modo, uma classe captura a semântica das características comuns a todas as suas instâncias.

Enquanto um objeto individual é uma entidade real, que executa algum papel no sistema como um todo, uma classe captura a estrutura e comportamento comum a todos os objetos que ela descreve. Assim, uma classe serve como uma espécie de contrato que deve ser estabelecido entre uma abstração e todos os seus clientes.
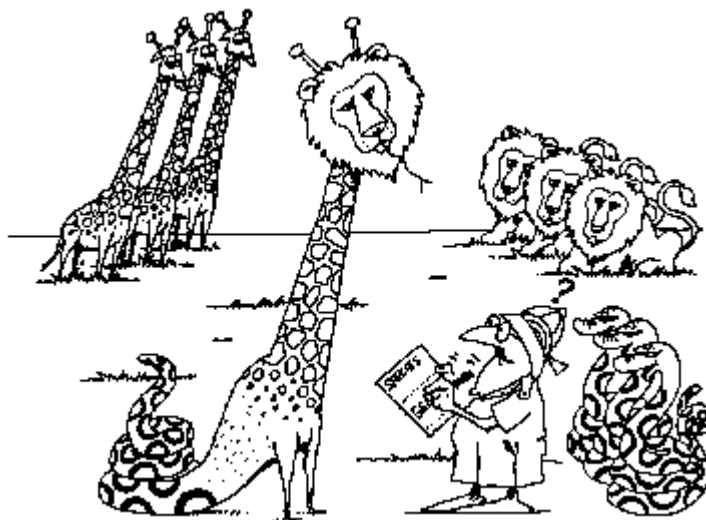
Figura 5.3 – Ilustração do conceito de Classificação (BOOCH, 1994).

Ligações e Associações

Em qualquer sistema, objetos relacionam-se uns com os outros. Por exemplo, em "*o empregado João trabalha no Departamento de Pessoal*", temos um relacionamento entre o objeto *empregado João* e o objeto *Departamento de Pessoal*.

Ligações e associações são meios de se representar relacionamentos entre objetos e entre classes, respectivamente. Uma ligação é uma conexão entre objetos. No exemplo anterior, há uma ligação entre os objetos *João* e *Departamento de Pessoal*. Uma associação, por sua vez, descreve um conjunto de ligações com estrutura e semântica comuns. No exemplo anterior, há uma associação entre as classes *Empregado* e *Departamento*. Todas as ligações de uma associação interligam objetos das mesmas classes e, assim, uma associação descreve um conjunto de potenciais ligações da mesma maneira que uma classe descreve um conjunto de potenciais objetos (BLAHA; RUMBAUGH, 2006).

Uma associação comum entre duas classes representa um relacionamento estrutural entre pares, significando que essas duas classes estão em um mesmo nível, sem que uma seja mais importante do que a outra. Além das associações comuns, a UML considera dois tipos de associações especiais entre objetos: *composição* e *agregação*. Ambos representam relações todo-parte. A agregação é uma forma especial de associação que especifica um relacionamento entre um objeto agregado (o todo) e seus componentes (as partes). A composição, por sua vez, é uma forma de agregação na qual o tempo de vida entre todo e partes é coincidente. As partes podem até ser criadas após a criação do todo, mas uma vez criadas, vivem e morrem com o todo. Uma parte pode ainda ser removida explicitamente antes da morte do todo (BOOCH; RUMBAUGH; JACOBSON, 2006).

Generalização / Especialização

Muitas vezes, um conceito geral pode ser especializado, adicionando-se novas características. Seja o exemplo do conceito de *estudante* no contexto de uma universidade. De modo geral, há características que são intrínsecas a quaisquer estudantes da universidade. Entretanto, é possível especializar este conceito para mostrar especificidades de subtipos de estudantes, tais como estudantes de graduação e estudantes de pós-graduação.

Da maneira inversa, pode-se extrair de um conjunto de conceitos, características comuns que, quando generalizadas, formam um conceito geral. Por exemplo, ao se avaliar os conceitos de carros, motos, caminhões e ônibus, pode-se notar que eles têm características comuns que podem ser generalizadas em um supertipo *veículo automotor terrestre*.

As abstrações de especialização e generalização são muito úteis para a estruturação de sistemas. Com elas, é possível construir hierarquias de classes. A *herança* é um mecanismo para modelar similaridades entre classes, representando as abstrações de generalização e especialização. Através da herança, é possível tornar explícitos propriedades e operações comuns em uma hierarquia de classes. O mecanismo de herança possibilita reutilização, captura explícita de características comuns e definição incremental de classes.

No que se refere à definição incremental de classes, a herança permite conceber uma nova classe como um refinamento de outras classes. A nova classe pode herdar as similaridades e definir apenas as novas características.

A herança é, portanto, um relacionamento entre classes (em contraposição às associações que representam relacionamentos entre objetos das classes), no qual uma classe compartilha a estrutura e comportamento definido em uma ou mais outras classes. A classe que herda características[1] é chamada *subclasse* e a que fornece as características, *superclasse*. Desta forma, a herança representa uma hierarquia de abstrações na qual uma subclasse herda de uma ou mais superclasses.

Tipicamente, uma subclasse aumenta ou redefine características de suas superclasses. Assim, se uma classe *B* herda de uma classe *A*, todas as características descritas em *A* tornam-se automaticamente parte de *B*, que ainda é livre para acrescentar novas características para seus propósitos específicos.

A generalização permite abstrair, a partir de um conjunto de classes, uma classe mais geral contendo todas as características comuns. A especialização é a operação inversa e, portanto, permite especializar uma classe em um número de subclasses, explicitando as diferenças entre as novas subclasses. Deste modo é possível compor a hierarquia de classes. Esses tipos de relacionamento são conhecidos também como relacionamentos "*é um tipo de*", onde um objeto da subclasse também "*é um tipo de*" objeto da superclasse. Neste caso uma instância da subclasse é dita uma *instância indireta* da superclasse.

Mensagens e Métodos

A abstração incorporada por um objeto é caracterizada por um conjunto de operações que podem ser requisitadas por outros objetos, ditos clientes. Métodos são implementações reais de operações. Para que um objeto realize alguma tarefa, é necessário enviar a ele uma mensagem, solicitando a execução de um método específico. Um cliente só pode acessar um objeto através da emissão de mensagens, isto é, ele não pode acessar ou manipular diretamente os dados associados ao objeto. Os objetos podem ser complexos e o cliente não precisa tomar conhecimento de sua complexidade interna. O cliente precisa saber apenas como se comunicar com o objeto e como ele reage. Assim, garante-se o encapsulamento.

As mensagens são o meio de comunicação entre objetos e são responsáveis pela ativação de todo e qualquer processamento. Dessa forma, é possível garantir que clientes não serão afetados por alterações nas implementações de um objeto que não alterem o comportamento esperado de seus serviços.

Classes e Operações Abstratas

Nem todas as classes são projetadas para instanciar objetos. Algumas são usadas simplesmente para organizar características comuns a diversas classes. Tais classes são ditas *classes abstratas*. Uma classe abstrata é desenvolvida basicamente para ser herdada por outras classes. Ela existe meramente para que um comportamento comum a um conjunto de classes possa ser colocado em uma localização comum e definido uma única vez. Assim, uma classe abstrata não possui instâncias diretas, mas suas classes descendentes *concretas*, sim. Uma classe concreta é uma classe instanciável, isto é, que pode ter instâncias diretas. Uma classe abstrata pode ter subclasses também abstratas, mas as classes-folhas na árvore de herança devem ser classes concretas.

[1] O termo *característica* é usado aqui para designar estrutura (atributos e associações) e comportamento (operações).

Classes abstratas podem ser projetadas de duas maneiras distintas. Primeiro, elas podem prover implementações completamente funcionais do comportamento que pretendem capturar. Alternativamente, elas podem prover apenas definição de um protocolo para uma operação sem apresentar um método correspondente. Tal operação é dita uma *operação genérica ou abstrata*. Neste caso, a classe abstrata não é completamente implementada e todas as suas subclasses concretas são obrigadas a prover uma implementação para suas operações abstratas. Assim, diz-se que uma operação abstrata define apenas a assinatura[2] a ser usada nas implementações que as subclasses deverão prover, garantindo, assim, uma interface consistente. Métodos que implementam uma operação genérica têm a mesma semântica.

Uma classe concreta não pode conter operações abstratas, porque senão seus objetos teriam operações indefinidas. Analogamente, toda classe que possuir uma operação genérica não pode ter instâncias diretas e, portanto, obrigatoriamente é uma classe abstrata.

5.3 - Levantamento e Registro de Requisitos

Uma vez que levantar requisitos não é tarefa fácil, é imprescindível que essa atividade seja cuidadosamente conduzida, fazendo uso de diversas técnicas. Dentre as diversas técnicas que podem ser aplicadas para o levantamento de requisitos, destacam-se: entrevistas, questionários, observação, investigação de documentos, prototipagem, cenários, abordagens em grupo, abordagens baseadas em objetivos e reutilização de requisitos.

Os resultados do levantamento de requisitos têm de ser registrados em um documento, de modo que possam ser verificados, validados e utilizados como base para outras atividades do processo de software. Para que sejam úteis, os requisitos têm de ser escritos em um formato compreensível por todos os interessados. Além disso, esses interessados devem interpretá-los uniformemente.

Normalmente, requisitos são documentados usando alguma combinação de linguagem natural, modelos, tabelas e outros elementos. A linguagem natural é quase sempre imprescindível, uma vez que é a forma básica de comunicação compreensível por todos os interessados. Contudo, ela geralmente abre espaços para ambiguidades e má interpretação. Assim, é interessante procurar estruturar o uso da linguagem natural e complementar a descrição dos requisitos com outros elementos.

Neste texto, sugerimos elaborar dois documentos: o Documento de Definição de Requisitos (ou somente **Documento de Requisitos**) e o **Documento de Especificação de Requisitos**. O Documento de Requisitos é mais sucinto, escrito em um nível mais apropriado para o cliente e contempla apenas os requisitos de usuário. O Documento de Especificação de Requisitos é mais detalhado, escrito a partir da perspectiva dos desenvolvedores (PFLEEGER, 2004), normalmente contendo diversos modelos para descrever requisitos de sistema.

Os requisitos de usuário devem ser descritos de modo a serem compreensíveis pelos interessados no sistema que não possuem conhecimento técnico detalhado. Eles devem especificar apenas o comportamento externo do sistema, em uma linguagem

[2] nome da operação, parâmetros e retorno.

simples, direta e sem usar terminologia específica de software (SOMMERVILLE, 2007).

5.3.1 O Documento de Requisitos

O Documento de Definição de Requisitos (ou simplesmente Documento de Requisitos) tem como propósito descrever os requisitos de usuário, tendo como público alvo clientes, usuários, gerentes (de cliente e de fornecedor) e desenvolvedores.

Há muitos formatos distintos propostos na literatura para documentos de requisitos. Neste texto, é proposta uma estrutura bastante simples para esse tipo de documento, contendo apenas quatro seções:

1. *Introdução:* breve introdução ao documento, descrevendo seu propósito e estrutura.
2. *Descrição do Propósito do Sistema:* descreve o propósito geral do sistema.
3. *Descrição do Minimundo:* apresenta, em um texto corrido, uma visão geral do domínio, do problema a ser resolvido e dos processos de negócio apoiados, bem como as principais ideias do cliente sobre o sistema a ser desenvolvido.
4. *Requisitos de Usuário:* apresenta os requisitos de usuário em linguagem natural. Três conjuntos de requisitos devem ser descritos nesta seção: requisitos funcionais, requisitos não funcionais e regras de negócio.

As três primeiras seções não têm nenhuma estrutura especial, sendo apresentadas na forma de um texto corrido. A introdução deve ser breve e basicamente descrever o propósito e a estrutura do documento, podendo seguir um padrão preestabelecido pela organização. A descrição do propósito do sistema deve ser direta e objetiva, tipicamente em um único parágrafo. Já a descrição do minimundo é um pouco maior, algo entre uma e duas páginas, descrevendo aspectos gerais e relevantes para um primeiro entendimento do domínio, do problema a ser resolvido e dos processos de negócio apoiados. Contém as principais ideias do cliente sobre o sistema a ser desenvolvido, obtidas no levantamento preliminar e exploratório do sistema. Não se devem incluir detalhes.

A seção 4, por sua vez, não deve ter um formato livre. Ao contrário, deve seguir um formato estabelecido pela organização, contendo, dentre outros: identificador do requisito, descrição, tipo, origem, prioridade, responsável, interessados, dependências em relação a outros requisitos e requisitos conflitantes. A definição de padrões organizacionais para a definição de requisitos é essencial para garantir uniformidade e evitar omissão de informações importantes acerca dos requisitos (SOMMERVILLE, 2007; WIEGERS, 2003). Como consequência, o padrão pode ser usado como um guia para a verificação de requisitos. A Tabela 5.1 apresenta o padrão tabular sugerido neste texto. Sugerem-se agrupar requisitos de um mesmo tipo em diferentes tabelas. Assim, a informação do tipo do requisito não aparece explicitamente no padrão proposto. Além disso, informações complementares podem ser adicionadas em função do tipo de requisito. A seguir, discute-se como cada um dos itens da tabela pode ser tratado, segundo uma perspectiva geral. Na sequência, são tecidas considerações mais específicas sobre a especificação dos diferentes tipos de requisitos.

Tabela 5.1 – Tabela de Requisitos.

Identificador	Descrição	Origem	Prioridade	Responsável	Interessados	Dependências	Conflitos

Os requisitos devem possuir identificadores únicos para permitir a identificação e o rastreamento na gerência de requisitos. Há diversas propostas de esquemas de rotulagem de requisitos. Neste texto, recomenda-se usar um esquema de numeração sequencial por tipo de requisito, sendo usados os seguintes prefixos para designar os diferentes tipos de requisitos: *RF – requisitos funcionais*; *RNF – requisitos não funcionais*; *RN – regras de negócio*. Para outros esquemas de rotulagem, vide (WIEGERS, 2003). É importante destacar que, quando um requisito é eliminado, seu identificador não pode ser atribuído a outro requisito.

A descrição do requisito normalmente é feita na forma de uma sentença em linguagem natural. Ainda que expressa em linguagem natural, é importante adotar um estilo consistente e usar a terminologia do usuário ao invés do jargão típico da computação. Em relação ao estilo, recomenda-se utilizar sentenças em um dos seguintes formatos para descrever requisitos funcionais e não funcionais:

- *O sistema deve <verbo indicando ação, seguido de complemento>*: use o verbo *dever* para designar uma função ou característica requerida para o sistema, ou seja, para descrever um requisito obrigatório. Exemplos: O sistema deve efetuar o controle dos clientes da empresa. O sistema deve processar um pedido do cliente em um tempo inferior a cinco segundos, contado a partir da entrada de dados.

- *O sistema pode <verbo indicando ação, seguido de complemento>*: use o verbo *poder* para designar uma função ou característica desejável para o sistema, ou seja, para descrever um requisito desejável, mas não essencial. Exemplos: O sistema pode notificar usuários em débito. O sistema pode sugerir outros produtos para compra, com base em produtos colocados no carrinho de compras do usuário.

Em algumas situações, pode-se querer explicitar que alguma funcionalidade ou característica não deve ser tratada pelo sistema. Neste caso, indica-se uma sentença com a seguinte estrutura: *O sistema não deve <verbo indicando ação, seguido de complemento>*.

Wiegers (2003) recomenda diversas diretrizes para a redação de requisitos, dentre elas:

- Escreva frases completas, com a gramática, ortografia e pontuação correta. Procure manter frases e parágrafos curtos e diretos.
- Use os termos consistentemente. Defina-os em um glossário.
- Prefira a voz ativa (o sistema deve fazer alguma coisa) à voz passiva (alguma coisa deve ser feita).
- Sempre que possível, identifique o tipo de usuário. Evite descrições genéricas como “o usuário deve [...]”. Se o usuário no caso for, por exemplo, o caixa do banco, indique claramente “o caixa do banco deve [...]”.

- Evite termos vagos, que conduzam a requisitos ambíguos e não testáveis, tais como “rápido”, “adequado”, “fácil de usar” etc.
- Escreva requisitos em um nível consistente de detalhe.
- O nível de especificação de um requisito deve ser tal que, se o requisito é satisfeito, a necessidade do cliente é atendida. Contudo, evite restringir desnecessariamente o projeto (*design*).
- Escreva requisitos individualmente testáveis. Um requisito bem escrito deve permitir a definição de um pequeno conjunto de testes para verificar se o requisito foi corretamente implementado.
- Evite longos parágrafos narrativos que contenham múltiplos requisitos. Divida um requisito desta natureza em vários menores.

Conforme apontado anteriormente, requisitos devem ser testáveis. Robertson e Robertson (2006) sugerem três maneiras de tornar os requisitos testáveis:

- Especificar uma descrição quantitativa de cada advérbio ou adjetivo, de modo que o significado dos qualificadores fique claro e não ambíguo.
- Trocar os pronomes pelos nomes das entidades.
- Garantir que todo nome (termo) importante seja definido em um glossário no documento de requisitos.

A origem de um requisito deve apontar a partir de que entidade (pessoa, documento, atividade) o requisito foi identificado. Um requisito identificado durante uma investigação de documentos, p.ex., tem como origem o(s) documento(s) inspecionado(s). Já um requisito levantado em uma entrevista com um certo usuário tem como origem o próprio usuário. A informação de origem é importante para se conseguir a rastrear requisitos para a sua origem, prática muito recomendada no contexto da gerência de requisitos.

Requisitos podem ter importância relativa diferente uns em relação aos outros. Assim, é importante que o cliente e outros interessados estabeleçam conjuntamente a prioridade de cada requisito.

É muito importante saber quem é o analista responsável por um requisito, bem como quem são os interessados (clientes, usuários etc.) naquele requisito. São eles que estarão envolvidos nas discussões relativas ao requisito, incluindo a tentativa de acabar com conflitos e a definição de prioridades. Assim, deve-se registrar o nome e o papel do responsável e dos interessados em cada requisito.

Um requisito pode depender de outros ou conflitar com outros. Quando as dependências e conflitos forem detectados, devem-se listar os respectivos identificadores nas colunas de dependências e conflitos.

Escrevendo Requisitos Funcionais

As diretrizes apresentadas anteriormente aplicam-se integralmente a requisitos funcionais. Assim, não há outras diretrizes específicas para os requisitos funcionais. Deve-se realçar apenas que, quando expressos como requisitos de usuário, requisitos funcionais são geralmente descritos de forma abstrata, não cabendo neste momento

entrar em detalhes. Detalhes vão ser naturalmente adicionados quando esses mesmos requisitos forem descritos na forma de requisitos de sistema.

Uma alternativa largamente empregada para especificar requisitos funcionais no nível de requisitos de sistema é a modelagem. Uma das técnicas mais comumente utilizadas para descrever requisitos funcionais como requisitos de sistema é a modelagem de casos de uso.

Escrevendo Requisitos Não Funcionais

Clientes e usuários naturalmente enfocam a especificação de requisitos funcionais. Entretanto para um sistema ser bem sucedido, é necessário mais do que entregar a funcionalidade correta. Usuários também têm expectativas sobre quão bem o sistema vai funcionar. Características que entram nessas expectativas incluem: quão fácil é usar o sistema, quão rapidamente ele roda, com que frequência ele falha e como ele trata condições inesperadas. Essas características, coletivamente conhecidas como atributos de qualidade do produto de software, são parte dos requisitos não funcionais do sistema. Essas expectativas de qualidade do produto têm de ser exploradas durante o levantamento de requisitos (WIEGERS, 2003).

Clientes geralmente não apontam suas expectativas de qualidade explicitamente. Contudo, informações providas por eles durante o levantamento de requisitos fornecem algumas pistas sobre o que eles têm em mente. Assim, é necessário definir com precisão o que os usuários pensam quando eles dizem que o sistema deve ser amigável, rápido, confiável ou robusto (WIEGERS, 2003).

Há muitos atributos de qualidade que podem ser importantes para um sistema. Uma boa estratégia para levantar requisitos não funcionais de produto consiste em explorar uma lista de potenciais atributos de qualidade que a grande maioria dos sistemas deve apresentar em algum nível. Por exemplo, o modelo de qualidade externa e interna de produtos de software definido na norma ISO/IEC 9126-1, utilizado como referência para a avaliação de produtos de software, define seis características de qualidade, desdobradas em subcaracterísticas, a saber (ISO/IEC, 2001):

- *Funcionalidade:* refere-se à existência de um conjunto de funções que satisfaz às necessidades explícitas e implícitas e suas propriedades específicas. Tem como subcaracterísticas: adequação, acurácia, interoperabilidade, segurança de acesso e conformidade.
- *Confiabilidade:* diz respeito à capacidade do software manter seu nível de desempenho, sob condições estabelecidas, por um período de tempo. Tem como subcaracterísticas: maturidade, tolerância a falhas, recuperabilidade e conformidade.
- *Usabilidade:* refere-se ao esforço necessário para se utilizar um produto de software, bem como o julgamento individual de tal uso por um conjunto de usuários. Tem como subcaracterísticas: inteligibilidade, apreensibilidade, operacionalidade, atratividade e conformidade.
- *Eficiência:* diz respeito ao relacionamento entre o nível de desempenho do software e a quantidade de recursos utilizados sob condições estabelecidas. Tem como subcaracterísticas: comportamento em relação ao tempo, comportamento em relação aos recursos e conformidade.

- *Manutenibilidade:* concerne ao esforço necessário para se fazer modificações no software. Tem como subcaracterísticas: analisabilidade, modificabilidade, estabilidade, testabilidade e conformidade.
- *Portabilidade:* refere-se à capacidade do software ser transferido de um ambiente para outro. Tem como subcaracterísticas: adaptabilidade, capacidade para ser instalado, coexistência, capacidade para substituir e conformidade.

É interessante observar que, mesmo dentre as subcaracterísticas de funcionalidade, há atributos que geralmente não são pensados pelos usuários como funções que o sistema deve prover, tais como interoperabilidade e segurança de acesso. Assim, é importante avaliar em que grau o sistema em questão necessita apresentar tais características.

Escrevendo Regras de Negócio

Toda organização opera de acordo com um extenso conjunto de políticas corporativas, leis, padrões industriais e regulamentações governamentais. Tais princípios de controle são coletivamente designados por regras de negócio. Uma regra de negócio é uma declaração que define ou restringe algum aspecto do negócio, com o propósito de estabelecer sua estrutura ou controlar ou influenciar o comportamento do negócio (WIEGERS, 2003).

Sistemas de informação tipicamente precisam fazer cumprir as regras de negócio. Ao contrário dos requisitos funcionais e não funcionais, a maioria das regras de negócio origina-se fora do contexto de um sistema específico. Assim, as regras a serem tratadas pelo sistema precisam ser identificadas, documentadas e associadas aos requisitos do sistema em questão (WIEGERS, 2003).

Wiegers identifica cinco tipos principais de regras de negócio, cada um deles apresentando uma forma típica de ser escrito:

- *Fatos ou invariantes:* declarações que são verdade sobre o negócio. Geralmente descrevem associações ou relacionamentos entre importantes termos do negócio. Ex.: Todo pedido tem uma taxa de remessa.
- *Restrições:* como o próprio nome indica, restringem as ações que o sistema ou seus usuários podem realizar. Algumas palavras ou frases sugerem a descrição de uma restrição, tais como *deve*, *não deve*, *não pode* e *somente*. Ex.: Um aluno só pode tomar emprestado, concomitantemente, de um a três livros.
- *Ativadores de Ações:* são regras que disparam alguma ação sob condições específicas. Uma declaração na forma “Se <alguma condição é verdadeira ou algum evento ocorre>, então <algo acontece>” é indicada para descrever ativadores de ações. Ex.: Se a data para retirada do livro é ultrapassada e o livro não é retirado, então a reserva é cancelada. Quando as condições que levam às ações são uma complexa combinação de múltiplas condições individuais, então o uso de tabelas de decisão ou árvores de decisão é indicado. As figuras 5.4 e 5.5 ilustram uma mesma regra de ativação de ações descrita por meio de uma árvore de decisão e de uma tabela de decisão, respectivamente.
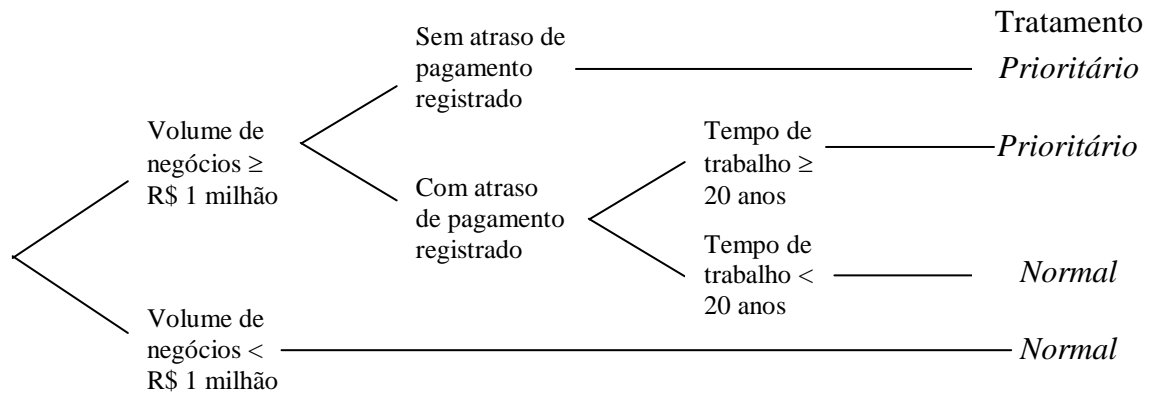
Figura 5.4 – Exemplo de Árvore de Decisão.

Tratamento de Clientes				
Volume de Negócios ≥ R$ 1 milhão?	S	S	S	N
Atraso de pagamento registrado?	N	S	S	-
Tempo de trabalho ≥ 20 anos?	-	S	N	-
Tratamento Prioritário	X	X		
Tratamento Normal			X	X

Figura 5.5 – Exemplo de Tabela de Decisão.

- *Inferências:* são regras que derivam novos fatos a partir de outros fatos ou cálculos. São normalmente escritas no padrão "se / então", como as regras ativadoras de ação, mas a cláusula então implica um fato ou nova informação e não uma ação a ser tomada. Ex.: Se o usuário não devolve um livro dentro do prazo estabelecido, então ele torna-se um usuário inadimplente.
- *Computações:* são regras de negócio que definem cálculos a serem realizados usando algoritmos ou fórmulas matemáticas específicos. Podem ser expressas como fórmulas matemáticas, descrição textual, tabelas, etc. Ex.: Aplica-se um desconto progressivo se mais do que 10 unidades forem adquiridas. De 10 a 19 unidades, o desconto é de 10%. A compra de 20 ou mais unidades tem um desconto de 25%. A Figura 5.6 mostra essa mesma regra expressa no formato de uma tabela.

Número de Itens Adquiridos	**Percentual de Desconto**
1 a 9	0
10 a 19	10%
20 ou mais	25%

Figura 5.6 – Exemplo de Regra de Cálculo.

Ao contrário de requisitos funcionais e não funcionais, regras de negócio não são passíveis de serem capturadas por meio de perguntas simples e diretas, tal como "Quais são suas regras de negócio?". Regras de negócio emergem durante a discussão de requisitos, sobretudo quando se procura entender a base lógica por detrás de requisitos e restrições apontados pelos interessados (WIEGERS, 2003). Assim, não se deve pensar que será possível levantar muitas regras de negócio em um levantamento preliminar de requisitos. Pelo contrário, as regras de negócio vão surgir principalmente durante o levantamento detalhado dos requisitos. Wiegers (2003) aponta diversas potenciais origens para regras de negócio e sugere tipos de questões que o analista pode fazer para tentar capturar regras advindas dessas origens:

- *Políticas:* Por que é necessário fazer isso desse jeito?
- *Regulamentações:* O que o governo requer?
- *Fórmulas:* Como este valor é calculado?
- *Modelos de Dados:* Como essas entidades de dados estão relacionadas?
- *Ciclo de Vida de Objetos:* O que causa uma mudança no estado desse objeto?
- *Decisões de Atores:* O que o usuário pode fazer a seguir?
- *Decisões de Sistema:* Como o sistema sabe o que fazer a seguir?
- *Eventos:* O que pode (e não pode) acontecer?

Regras de negócio normalmente têm estreita relação com requisitos funcionais. Uma regra de negócio pode ser tratada no contexto de uma certa funcionalidade. Neste caso, a regra de negócio deve ser listada na coluna de dependências do requisito funcional (vide Tabela 5.1). Há casos em que uma regra de negócio conduz a um requisito funcional para fazer cumprir a regra. Neste caso, a regra de negócio é considerada a origem do requisito funcional (WIEGERS, 2003).

É importante destacar a importância das regras (restrições) obtidas a partir de modelos de dados, ditas *restrições de integridade*. Elas complementam as informações do modelo de dados e capturam restrições entre elementos de um modelo de dados que não são passíveis de serem capturadas pelas notações gráficas utilizadas em modelos desse tipo. Tais regras devem ser documentadas junto com os modelos de dados. Seja o exemplo do modelo de dados da Figura 5.7. Esse fragmento de modelo indica que: (i) um aluno cursa um curso; (ii) um aluno pode se matricular em nenhuma ou várias turmas; (iii) um curso possui um conjunto de disciplinas em sua matriz curricular; (iv) uma turma é de uma disciplina específica. Contudo, nada diz sobre restrições entre o estabelecimento dessas várias relações. Suponha que o negócio indique que a seguinte restrição deve ser considerada: Um aluno só pode ser matricular em turmas de disciplinas que compõem a grade curricular do curso que esse aluno cursa. Essa restrição tem de ser escrita para complementar o modelo.
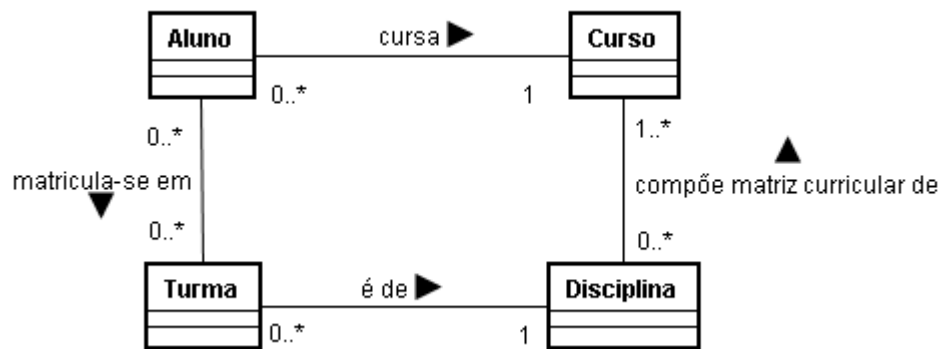
Figura 5.7 – Exemplo de Fragmento de Modelo de Dados com Restrição de Integridade.

Outro tipo de restrição importante são as regras que impõem restrições sobre funcionalidades de inserção, atualização ou exclusão de dados, devido a relacionamentos de existentes entre as entidades. Voltando ao exemplo da Figura 5.7, a exclusão de disciplinas não deve ser livre, uma vez que turmas são existencialmente dependentes de disciplinas. Assim, a seguinte regra de integridade referencial de dados deve ser considerada: Ao excluir uma disciplina, devem-se excluir todas as turmas a ela relacionadas. Essas regras são denominadas neste texto como *restrições de processamento* e, uma vez que dizem respeito a funcionalidades, devem ser documentadas junto com a descrição do caso de uso que detalha a respectiva funcionalidade.

5.3.3 – O Documento de Especificação de Requisitos

Os requisitos de sistema, assim como foi o caso dos requisitos de usuário, têm de ser especificados em um documento, de modo a poderem ser verificados e validados e posteriormente usados como base para as atividades subsequentes do desenvolvimento de software. O Documento de Especificação de Requisitos tem como propósito registrar os requisitos escritos a partir da perspectiva do desenvolvedor e, portanto, deve incluir os vários modelos conceituais desenvolvidos, bem como a especificação dos requisitos não funcionais detalhados usando critérios de aceitação ou cenários de atributos de qualidade.

Diferentes formatos podem ser propostos para documentos de especificação requisitos, bem como mais de um documento pode ser usado para documentar os requisitos de sistema. Neste texto, propõe-se o uso de um único documento, contendo as seguintes informações:

1. *Introdução:* breve introdução ao documento, descrevendo seu propósito e estrutura.
2. *Modelo de Casos de Uso:* apresenta o modelo de casos de uso do sistema, incluindo os diagramas de casos de uso e as descrições de casos de uso associadas.
3. *Modelo Estrutural:* apresenta o modelo estrutural do sistema, incluindo os diagramas de classes do sistema.

4. *Modelo Dinâmico:* apresenta os modelos comportamentais dinâmicos do sistema, incluindo os diagramas de estados, diagramas de interação e diagramas de atividades[3].

5. *Dicionário do Projeto:* apresenta as definições dos principais conceitos capturados pelos diversos modelos e restrições de integridade a serem consideradas, servindo como um glossário do projeto.

6. *Especificação dos Requisitos Não Funcionais:* apresenta os requisitos não funcionais descritos no nível de sistema, o que inclui critérios de aceitação e cenários de atributos de qualidade.

É importante frisar que dificilmente um sistema é simples o bastante para ser modelado como um todo. Quase sempre é útil dividir um sistema em unidades menores, mais fáceis de serem gerenciáveis, ditas subsistemas. É útil organizar a especificação de requisitos por subsistemas e, portanto, cada uma das seções propostas acima pode ser subdividida por subsistemas.

Um modelo estrutural para uma aplicação complexa, por exemplo, pode ter centenas de classes e, portanto, pode ser necessário definir uma representação concisa capaz de orientar um leitor em um modelo dessa natureza. O agrupamento de elementos de modelo em subsistemas serve basicamente a este propósito, podendo ser útil também para a organização de grupos de trabalho em projetos extensos. A base principal para a identificação de subsistemas é a complexidade do domínio do problema. Através da identificação e agrupamento de elementos de modelo em subsistemas, é possível controlar a visibilidade do leitor e, assim, tornar os modelos mais compreensíveis.

A UML (*Unified Modeling Language*)[4] provê um tipo principal de item de agrupamento, denominado pacote, que é um mecanismo de propósito geral para a organização de elementos da modelagem em grupos. Um diagrama de pacotes mostra a decomposição de um modelo em unidades menores e suas dependências, como ilustra a Figura 5.8. A linha pontilhada direcionada indica que o pacote origem (no exemplo, o pacote Atendimento a Cliente) depende do pacote destino (no exemplo, o pacote Controle de Acervo).
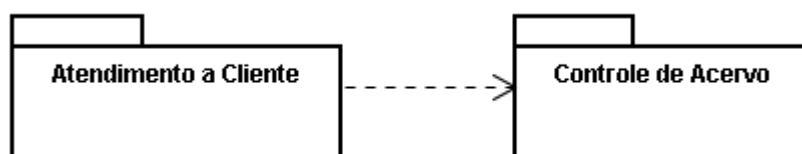
Figura 5.8 – Exemplo de um Diagrama de Pacotes.

Nas próximas seções são realizadas as discussões necessárias para que sejam elaborados os Modelos de Casos de Uso, os Modelos Estruturais e os Diagramas de Estados (modelo dinâmico).

---

[3] Neste material, no que diz respeito à modelagem dinâmica, serão abordados apenas os diagramas de estados.

[4] A UML é uma linguagem gráfica padrão para especificar, visualizar, documentar e construir artefatos de sistemas de software (BOOCH; RUMBAUGH; JACOBSON, 2006). Tipicamente, é a linguagem utilizada na criação dos modelos gerados durante as etapas de especificação e análise de requisitos e projeto de sistema.

5.4 Modelagem de Casos de Uso

Modelos de caso de uso (*use cases*) são modelos passíveis de compreensão tanto por desenvolvedores – analistas, projetistas, programadores e testadores – como pela comunidade usuária – clientes e usuários. Como o próprio nome sugere, um caso de uso é uma maneira de usar o sistema. Usuários interagem com o sistema, interagindo com seus casos de uso. Tomados em conjunto, os casos de uso de um sistema representam a sua funcionalidade. Casos de uso são, portanto, os "itens" que o desenvolvedor negocia com seus clientes.

O propósito do modelo de casos de uso é capturar e descrever a funcionalidade que um sistema deve prover. Um sistema geralmente serve a vários atores, para os quais ele provê diferentes serviços. Tipicamente, a funcionalidade a ser provida por um sistema é muito grande para ser analisada como uma única unidade e, portanto, é importante ter um mecanismo de dividir essa funcionalidade em partes menores e mais gerenciáveis. O conceito de caso de uso é muito útil para esse propósito (OLIVÉ, 2007).

É importante ter em mente que casos de uso são fundamentalmente uma ferramenta textual. Ainda que casos de uso sejam também descritos graficamente (p.ex., fluxogramas ou algum diagrama da UML, dentre eles diagramas de casos de uso, diagramas de sequência e diagramas de atividades), não se deve perder de vista a natureza textual dos casos de uso. Olhando casos de uso apenas a partir da UML, que não trata do conteúdo ou da escrita de casos de uso, pode-se pensar, equivocadamente, que casos de uso são uma construção gráfica ao invés de textual.

Em essência, casos de uso servem como um meio de comunicação entre pessoas, algumas delas sem nenhum treinamento especial e, portanto, o uso de texto para especificar casos de uso é geralmente a melhor escolha. Casos de uso são amplamente usados no desenvolvimento de sistemas, porque, por meio sobretudo de suas descrições textuais, usuários e clientes conseguem visualizar qual a funcionalidade a ser provida pelo sistema, conseguindo reagir mais rapidamente no sentido de refinar, alterar ou rejeitar as funções previstas para o sistema (COCKBURN, 2005). Assim, um modelo de casos de uso inclui duas partes principais: (i) os diagramas de casos de uso e (ii) as descrições de atores e de casos de uso, sendo que essas últimas podem ser complementadas com outros diagramas associados, tais como os diagramas de atividade e de sequência da UML.

Outro aspecto a ser realçado é que os modelos de caso de uso são independentes do método de análise a ser usado e até mesmo do paradigma de desenvolvimento. Assim, pode-se utilizar a modelagem de casos de uso tanto no contexto do desenvolvimento orientado a objetos (foco deste texto), como em projetos desenvolvidos segundo o paradigma estruturado. De fato, o uso de modelos de caso de uso pode ser ainda mais amplo. Casos de uso podem ser usados, por exemplo, para documentar processos de negócio de uma organização. Contudo, neste texto, explora-se a utilização de casos de uso para modelar e documentar requisitos funcionais de sistemas. Assim, geralmente são interessados[5] (*stakeholders*) nos casos de uso: as pessoas que usarão o sistema (usuários), o cliente que requer o sistema, outros sistemas com os quais o sistema em questão terá de interagir e outros membros da organização (ou até mesmo de fora dela) que têm restrições que o sistema precisa garantir.

[5] Alguém ou algo com interesse no comportamento do sistema sob discussão (COCKBURN, 2005).

5.4.1 Atores

Dá-se nome de ator a um papel desempenhado por entidades físicas (pessoas, organizações ou outros sistemas) que interagem com o sistema em questão da mesma maneira, procurando atingir os mesmos objetivos. Uma mesma entidade física pode desempenhar diferentes papéis no mesmo sistema, bem como um dado papel pode ser desempenhado por diferentes entidades (OLIVÉ, 2007).

Atores são externos ao sistema. Um ator se comunica diretamente com o sistema, mas não é parte dele. A modelagem dos atores ajuda a definir as fronteiras do sistema, isto é, o conjunto de atores de um sistema delimita o ambiente externo desse sistema, representando o conjunto completo de entidades para as quais o sistema pode servir (BLAHA; RUMBAUGH, 2006; OLIVÉ, 2007).

Uma dúvida que sempre passa pela cabeça de um iniciante em modelagem de casos de uso é saber se o ator é a pessoa que efetivamente opera o sistema (p.ex., o atendente de uma locadora de automóveis) ou se é a pessoa interessada no resultado do processo (p.ex., o cliente que efetivamente loca o automóvel e é atendido pelo atendente). Essa definição depende, em essência, da fronteira estabelecida para o sistema. Sistemas de informação podem ter diferentes níveis de automatização. Por exemplo, se um sistema roda na Internet, seu nível de automatização é maior do que se ele requer um operador. Assim, é importante capturar qual o nível de automatização requerido e levar em conta o real limite do sistema (WAZLAWICK, 2004). Se o caso de uso roda na Internet (p.ex., um caso de uso de reserva de automóvel), então o cliente é o ator efetivamente. Se o caso de uso requer um operador (p.ex., um caso de uso de locação de automóvel, disponível apenas na locadora e para ser usado por atendentes), então o operador é o ator.

Quando se for considerar um sistema como sendo um ator, deve-se tomar o cuidado para não confundir a ideia de sistema externo (ator) com produtos usados na implementação do sistema em desenvolvimento. Para que um sistema possa ser considerado um ator, ele deve ser um sistema de informação completo (e não apenas uma biblioteca de classes, por exemplo). Além disso, ele deve estar fora do escopo do desenvolvimento do sistema atual. O analista não terá a oportunidade de alterar as funções do sistema externo, devendo adequar a comunicação às características do mesmo (WAZLAWICK, 2004).

Um ator primário é um ator que possui metas a serem cumpridas através do uso de serviços do sistema e que, tipicamente, inicia a interação com o sistema (OLIVÉ, 2007). Um ator secundário é um ator externo que interage com o sistema para prover um serviço para este último. A identificação de atores secundários é importante, uma vez que ela permite identificar interfaces externas que o sistema usará e os protocolos que regem as interações ocorrendo através delas (COCKBURN, 2005).

De maneira geral, o ator primário é o usuário direto do sistema ou outro sistema computacional que requisita um serviço do sistema em desenvolvimento. O sistema responde à requisição procurando atendê-la, ao mesmo tempo em que protege os interesses de todos os demais interessados no caso de uso. Entretanto, há situações em que o iniciador do caso de uso não é o ator primário. O tempo, por exemplo, pode ser o acionador de um caso de uso. Um caso de uso que roda todo dia à meia noite ou ao final do mês tem o tempo como acionador. Mas o caso de uso ainda visa atingir um objetivo de um ator e esse ator é considerado o ator primário do caso de uso, ainda que ele não interaja efetivamente com o sistema (COCKBURN, 2005).

Para nomear atores, recomenda-se o uso de substantivos no singular, iniciados com letra maiúscula, possivelmente combinados com adjetivos. Exemplos: Cliente, Bibliotecário, Correntista, Correntista Titular etc.

5.4.2 – Casos de Uso

Um caso de uso é uma porção coerente da funcionalidade que um sistema pode fornecer para atores interagindo com ele (BLAHA; RUMBAUGH, 2006). Um caso de uso corresponde a um conjunto de ações realizadas pelo sistema (ou por meio da interação com o sistema), que produz um resultado observável, com valor para um ou mais atores do sistema. Geralmente, esse valor é a realização de uma meta de negócio ou tarefa (OLIVÉ, 2007). Assim, um caso de uso captura alguma função visível ao ator e, em especial, busca atingir uma meta desse ator.

Deve-se considerar que um caso de uso corresponde a uma transação completa, ou seja, um usuário poderia ativar o sistema, executar o caso de uso e desativar o sistema logo em seguida, e a operação estaria completa e consistente e atenderia a uma meta desse usuário (WAZLAWICK, 2004).

Ser uma transação completa é uma característica essencial de um caso de uso[6], pois somente transações completas são capazes de atingir um objetivo do usuário. Casos de uso que necessitam de múltiplas seções não passam nesse critério e devem ser divididos em casos de uso menores. Seja o exemplo de um caso de uso de concessão de empréstimo. Inicialmente, um atendente interagindo com um cliente informa os dados necessários para a avaliação do pedido de empréstimo. O pedido de empréstimo é, então, enviado para análise por um analista de crédito. Uma vez analisado e aprovado, o empréstimo é concedido, quando o dinheiro é entregue ao cliente e um contrato é assinado, dentre outros. Esse processo pode levar vários dias e não é realizado em uma seção única. Assim, o caso de uso de concessão de empréstimo deveria ser subdividido em casos de uso menores, tais como casos de uso para efetuar pedido de empréstimo, analisar pedido de empréstimo e formalizar concessão de empréstimo.

Por outro lado, casos de uso muito pequenos, que não caracterizam uma transação completa, devem ser considerados passos de um caso de uso maior[7]. Seja o exemplo de uma biblioteca a qual cobra multa na devolução de livros em atraso. Um caso de uso específico para apenas calcular o valor da multa não é relevante, pois não caracteriza uma transação completa capaz de atingir um objetivo do usuário. O objetivo do usuário é efetuar a devolução e, neste contexto, uma regra de negócio (a que estabelece a multa) tem de ser levada em conta. Assim, calcular a multa é apenas um passo do caso de uso que efetua a devolução, o qual captura uma ação do sistema para garantir a regra de negócio e, portanto, satisfazer um interesse da biblioteca como organização.

Um caso de uso reúne todo o comportamento relevante de uma parte da funcionalidade do sistema. Isso inclui o comportamento principal normal, as variações de comportamento normais, as condições de exceção e o cancelamento de uma requisição. O conjunto de casos de uso captura a funcionalidade completa do sistema (BLAHA; RUMBAUGH, 2006).

---

[6] Esta regra tem como exceção os casos de uso de inclusão e extensão, conforme discutido mais adiante na seção que trata de relacionamentos entre casos de uso.
[7] As mesmas exceções da nota anterior se aplicam aqui, conforme discutido mais adiante.

Casos de uso fornecem uma abordagem para os desenvolvedores chegarem a uma compreensão comum com os usuários finais e especialistas do domínio, acerca da funcionalidade a ser provida pelo sistema (BOOCH; RUMBAUGH; JACOBSON, 2006).

Os objetivos dos atores são um bom ponto de partida para a identificação de casos de uso. Pode-se propor um caso de uso para satisfazer cada um dos objetivos de cada um dos atores. A partir desses objetivos, podem-se estudar as possíveis interações do ator com o sistema e refinar o modelo de casos de uso.

Cada caso de uso tem um nome. Esse nome deve capturar a essência do caso de uso. Para nomear casos de uso sugere-se usar frases iniciadas com verbos no infinitivo, seguidos de complementos, que representem a meta ou tarefa a ser realizada com o caso de uso. As primeiras letras (exceto preposições) de cada palavra devem ser grafadas em letra maiúscula. Exemplos: Cadastrar Cliente, Devolver Livro, Efetuar Pagamento de Fatura etc.

Um caso de uso pode ser visto como um tipo cujas instâncias são *cenários*. Um cenário é uma execução de um caso de uso com entidades físicas particulares desempenhando os papéis dos atores e em um particular estado do domínio de informação. Um cenário, portanto, exercita um certo caminho dentro do conjunto de ações de um caso de uso (OLIVÉ, 2007).

Alguns cenários mostram o objetivo do caso de uso sendo alcançado; outros terminam com o caso de uso sendo abandonado (COCKBURN, 2005). Mesmo quando o objetivo de um caso de uso é alcançado, ele pode ser atingido seguindo diferentes caminhos. Assim, um caso de uso deve comportar todas essas situações. Para tal, um caso de uso é normalmente descrito por um conjunto de fluxos de eventos, capturando o fluxo de eventos principal, i.e., o fluxo de eventos típico que conduz ao objetivo do caso de uso, e fluxos de eventos alternativos, descrevendo exceções ou variantes do fluxo principal.

5.4.3 - Diagramas de Casos de Uso

Basicamente, um diagrama de casos de uso mostra um conjunto de casos de uso e atores e seus relacionamentos, sendo utilizado para ilustrar uma visão estática das maneiras possíveis de se usar o sistema (BOOCH; RUMBAUGH; JACOBSON, 2006).

Os diagramas de casos de uso da UML podem conter os seguintes elementos de modelo, ilustrados na Figura 5.9 (BOOCH; RUMBAUGH; JACOBSON, 2006):

- *Assunto:* o assunto delimita a fronteira de um diagrama de casos de uso, sendo normalmente o sistema ou um subsistema. Os casos de uso de um assunto descrevem o comportamento completo do assunto. O assunto é exibido em um diagrama de casos de uso como um retângulo envolvendo os casos de uso que o compõem. O nome do assunto (sistema ou subsistema) pode ser mostrado dentro do retângulo.
- *Ator:* representa um conjunto coerente de papéis que os usuários ou outros sistemas desempenham quando interagem com os casos de uso. Tipicamente, um ator representa um papel que um ser humano, um dispositivo de hardware ou outro sistema desempenha com o sistema em questão. Atores não são parte do sistema. Eles residem fora do sistema. Atores são

representados por um ícone de homem, com o nome colocado abaixo do ícone.

- *Caso de Uso:* representa uma funcionalidade que o sistema deve prover. Casos de uso são parte do sistema e, portanto, residem dentro dele. Um caso de uso é representado por uma elipse com o nome do caso de uso dentro ou abaixo dela.
- *Relacionamentos de Dependência, Generalização e Associação*: são usados para estabelecer relacionamentos entre atores, entre atores e casos de uso, e entre casos de uso.
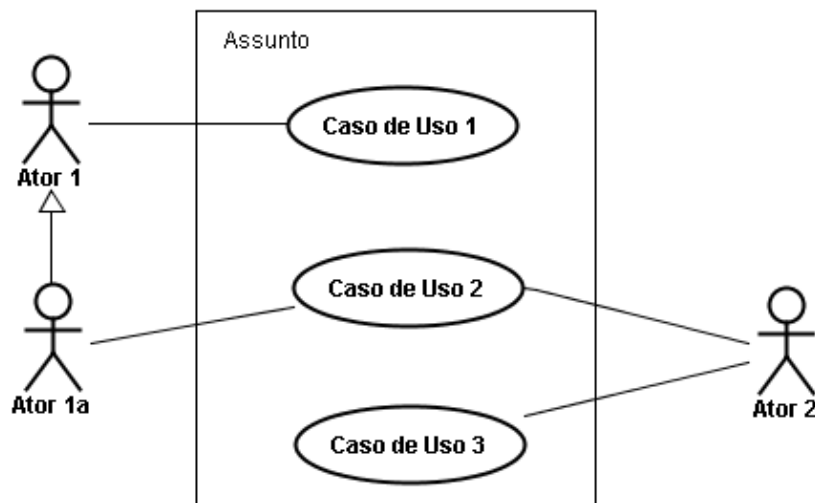
Figura 5.9 - Diagrama de Casos de Uso – Conceitos e Notação.

Atores só podem estar conectados a casos de uso por meio de *associações*. Uma associação entre um ator e um caso de uso significa que estímulos podem ser enviados entre atores e casos de uso. A associação entre um ator e um caso de uso indica que o ator e o caso de uso se comunicam entre si, cada um com a possibilidade de enviar e receber mensagens (BOOCH; RUMBAUGH; JACOBSON, 2006).

Atores podem ser organizados em hierarquias de generalização / especialização, de modo a capturar que um ator filho herda o significado e as associações com casos de uso de seu pai, especializando esse significado e potencialmente adicionando outras associações como outros casos de uso.

A Figura 5.10 mostra um diagrama de casos de uso para um sistema de caixa automático. Nesse diagrama, o assunto é o sistema como um todo. Os atores são: os clientes do banco, o sistema bancário e os responsáveis pela manutenção do numerário no caixa eletrônico. Cliente e mantenedor são atores primários, uma vez que têm objetivos a serem atingidos pelo uso do sistema. O sistema bancário é um ator, pois o sistema do caixa automático precisa interagir com o sistema bancário para realizar os casos de uso *Efetuar Saque*, *Emitir Extrato* e *Efetuar Pagamento*.
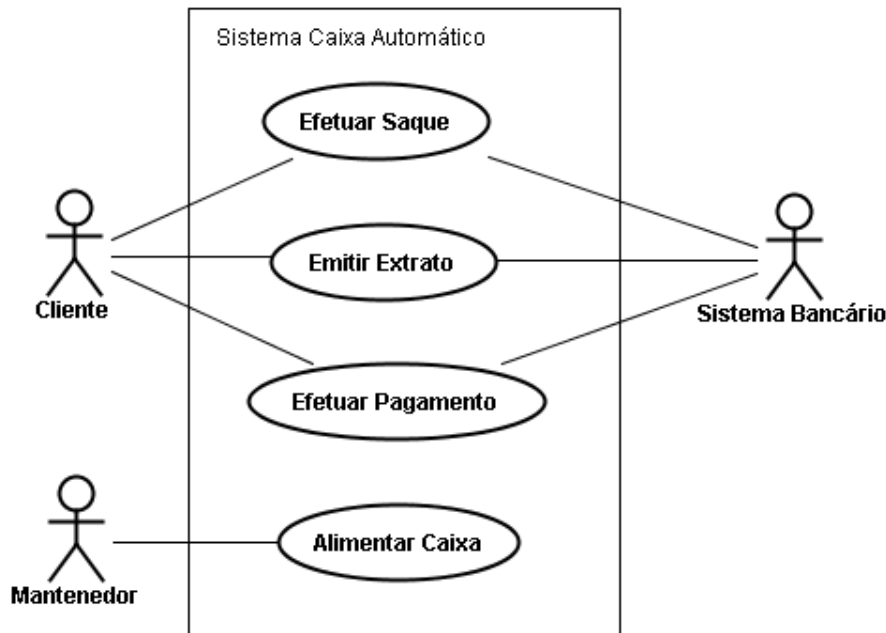
Figura 5.10 - Diagrama de Casos de Uso – Caixa Automático.

Um caso de uso descreve o que um sistema deve fazer. O diagrama de casos de uso provê uma visão apenas parcial disso, uma vez que mostra as funcionalidades por perspectiva externa. É necessário, ainda, capturar uma visão interna de cada caso de uso, especificando o comportamento do caso de uso pela descrição do fluxo de eventos que ocorre internamente (passos do caso de uso). Assim, uma parte fundamental do modelo de casos de uso é a descrição dos casos de uso.

5.4.4 - Descrevendo Casos de Uso

Um caso de uso deve descrever *o que* um sistema faz. Exceto para situações muito simples, um diagrama de casos de uso é insuficiente para este propósito. Assim, deve-se especificar o comportamento de um caso de uso pela descrição textual de seu fluxo de eventos, de modo que outros interessados possam compreendê-lo.

A especificação ou descrição de um caso de uso deve conter, dentre outras informações, um conjunto de sentenças, cada uma delas escrita em uma forma gramatical designando um passo simples, de modo que aprender a ler um caso de uso não requeira mais do que uns poucos minutos. Dependendo da situação, diferentes estilos de escrita podem ser adotados (COCKBURN, 2005).

Cada passo do fluxo de eventos de um caso de uso tipicamente descreve uma das seguintes situações: (i) uma interação entre um ator e o sistema, (ii) uma ação que o sistema realiza para atingir o objetivo do ator primário ou (iii) uma ação que o sistema realiza para proteger os interesses de um interessado. Essas ações podem incluir validações e mudanças do estado interno do sistema (COCKBURN, 2005).

Não há um padrão definido para especificar casos de uso. Diferentes autores propõem diferentes estruturas, formatos e conteúdos para descrições de casos de uso, alguns mais indicados para casos de uso essenciais e mais complexos, outros para casos de uso cadastrais e mais simples. Mais além, pode ser útil utilizar mais de um formato dentro do mesmo projeto, em função das peculiaridades de cada caso de uso. De todo

modo, é recomendável que a organização defina modelos de descrição de casos de uso a serem adotados em seus projetos, devendo definir tantos modelos quantos julgar necessários.

As seguintes informações são um bom ponto de partida para a definição de um modelo de descrição de casos de uso:

- *Nome:* nome do caso de uso, capturando a sua essência
- *Escopo:* diz respeito ao que está sendo documentado pelo caso de uso. Tipicamente pode ser um processo de negócio, um sistema ou um subsistema. Vale lembrar que este texto não aborda a utilização de casos de uso para a modelagem de processos de negócio. Assim, o escopo vai apontar o sistema / subsistema do qual o caso de uso faz parte.
- *Descrição do Propósito:* uma descrição sucinta do caso de uso, na forma de um único parágrafo, procurando descrever o objetivo do caso de uso.
- *Ator Primário:* nome do ator primário, ou seja, o interessado que tem um objetivo em relação ao sistema, o qual pode ser atingido pela execução do caso de uso.
- *Interessados e Interesses:* um interessado é alguém ou algo (um outro sistema) que tem um interesse no comportamento do caso de uso sendo descrito. Nesta seção são descritos cada um dos interessados no sistema e qual o seu interesse no caso de uso, incluindo o ator primário.
- *Pré-condições:* o que deve ser verdadeiro antes da execução do caso de uso. Se as pré-condições não forem satisfeitas, o caso de uso não pode ser realizado.
- *Pós-condições:* o que deve ser verdadeiro após a execução do caso de uso, considerando que o fluxo de eventos normal é realizado com sucesso.
- *Fluxo de Eventos Normal:* descreve os passos do caso de uso realizados em situações normais, considerando que nada acontece de errado e levando em conta a maneira mais comum do caso de uso ser realizado.
- *Fluxo de Eventos Alternativos:* descreve formas alternativas de realizar certos passos do caso de uso. Há duas formas alternativas principais: fluxos variantes, que são considerados dentro da normalidade do caso de uso; e fluxos de exceção, que se referem ao tratamento de erros durante a execução de um passo do fluxo normal (ou de um fluxo variante ou até mesmo de um outro fluxo de exceção).
- *Requisitos Relacionados:* listagem dos identificadores dos requisitos (funcionais, não funcionais e regras de negócio) tratados pelo caso de uso sendo descrito, de modo a permitir rastrear os requisitos. Casos de uso podem ser usados para conectar vários requisitos, de tipos diferentes. Assim, essa listagem ajuda a manter um rastro entre requisitos funcionais, não funcionais e regras de negócio, além de permitir verificar se algum requisito deixou de ser tratado.
- *Classes / Entidades:* classes (no paradigma orientado a objetos) ou entidades (no paradigma estruturado) necessárias para tratar o caso de uso sendo descrito. Esta seção é normalmente preenchida durante a modelagem

conceitual estrutural e é igualmente importante para permitir rastrear requisitos para as etapas subsequentes do desenvolvimento (projeto e implementação, sobretudo).

5.4.5 – Descrevendo os Fluxos de Eventos

Uma vez que o conjunto inicial de casos de uso estiver estabilizado, cada um deles deve ser descrito em mais detalhes. Primeiro, deve-se descrever o fluxo de eventos principal (ou curso básico), isto é, o curso de eventos mais importante, que normalmente ocorre. O fluxo de eventos normal (ou principal) é uma informação essencial na descrição de um caso de uso e não pode ser omitido em nenhuma circunstância. O fluxo de eventos normal é, portanto, a principal seção de uma descrição de caso de uso, a qual descreve o processo quando tudo dá certo, ou seja, sem a ocorrência de nenhuma exceção (WAZLAWICK, 2004).

Variantes do curso básico de eventos e tratamento de exceções que possam vir a ocorrer devem ser descritos em cursos alternativos. Normalmente, um caso de uso possui apenas um único curso básico, mas diversos cursos alternativos. Seja o exemplo de um sistema de caixa automático de banco, cujo diagrama de casos de uso é mostrado na Figura 5.10. O caso de uso *Efetuar Saque* poderia ser descrito como mostrado na Figura 5.11.

Como visto nesse exemplo, um caso de uso pode ter um número de cursos alternativos que podem levar o caso de uso por diferentes caminhos. Tanto quanto possível, esses cursos alternativos, muitos deles cursos de exceção, devem ser identificados durante a especificação do fluxo de eventos normal de um caso do uso.

Vale realçar que uma exceção não é necessariamente um evento que ocorre muito raramente, mas sim um evento capaz de impedir o prosseguimento do caso de uso, se não for devidamente tratado. Uma exceção também não é algo que impede o caso de uso de ser iniciado, mas algo que impede a sua conclusão. Condições que impedem um caso de uso de ser iniciado devem ser tratadas como pré-condições. As pré-condições nunca devem ser testadas durante o processo do caso de uso, pois, por definição, elas impedem que o caso de uso seja iniciado. Logo, seria inconsistente imaginar que elas pudessem ocorrer durante a execução do caso de uso. Se uma pré-condição é falsa, então o caso de uso não pode ser iniciado (WAZLAWICK, 2004).

Observa-se que a maioria das exceções ocorre nos passos em que alguma informação é passada dos atores para o sistema. Isso porque, quando uma informação é passada para o sistema, muitas vezes ele realiza validações. Quando uma dessas validações falha, tipicamente ocorre uma exceção (WAZLAWICK, 2004).

Nome: Efetuar Saque

Escopo: Sistema de Caixa Automático

Descrição do Propósito: Este caso de uso permite que um cliente do banco efetue um saque, retirando dinheiro de sua conta bancária.

Ator Primário: Cliente
Interessados e Interesses:

- Cliente: deseja efetuar um saque.
- Banco: garantir que apenas o próprio cliente efetuará saques e que os valores dos saques sejam compatíveis com o limite de crédito do cliente.

Pré-condições: O caixa automático deve estar conectado ao sistema bancário.

Pós-condições: O saque é efetuado, debitando o valor da conta do cliente e entregando o mesmo valor para o cliente em espécie.

Fluxo de Eventos Normal

O cliente insere seu cartão no caixa automático, que analisa o cartão e verifica se ele é aceitável. Se o cartão é aceitável, o caixa automático solicita que o cliente informe a senha. O cliente informa a senha. O caixa automático envia os dados do cartão e da senha para o sistema bancário para validação. Se a senha estiver correta, o caixa solicita que o cliente informe o tipo de transação a ser efetuada. O cliente seleciona a opção saque e o caixa solicita que seja informada a quantia. O cliente informa a quantia a ser sacada. O caixa envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada. Se o saque é autorizado, as notas são preparadas e liberadas.

Fluxos de Eventos de Exceção

- O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada.
- Senha incorreta: Se a senha informada está incorreta, uma mensagem é mostrada para o cliente que poderá entrar com a senha novamente. Caso o cliente informe três vezes senha incorreta, o cartão deverá ser bloqueado.
- Saque não autorizado: Se o saque não for aceito pelo sistema bancário, uma mensagem de erro é exibida e a operação é abortada.
- Não há dinheiro suficiente disponível no caixa eletrônico: Uma mensagem de erro é exibida e a operação é abortada.
- Cancelamento: O cliente pode cancelar a transação a qualquer momento, enquanto o saque não for autorizado pelo sistema bancário.

Requisitos Relacionados: RF01, RN01, RNF01, RNF02[8]

Classes: Cliente, Conta, Cartão, Transação, Saque.

Figura 5.11 – Descrição do Caso de Uso *Efetuar Saque*.

[8] São as seguintes as descrições dos requisitos listados: RF01 – O sistema de caixa automático deve permitir que clientes efetuem saques em dinheiro; RN01 – Não devem ser permitidas transações que deixem a conta do cliente com saldo inferior ao de seu limite de crédito; RNF01 – O sistema de caixa automático deve estar integrado ao sistema bancário; RNF02 – As operações realizadas no caixa automático devem dar respostas em até 10s a partir da entrada de dados.

Em sistemas de médio a grande porte, pode ser útil considerar a fusão de casos de uso fortemente relacionados em um único caso de uso, contendo mais de um fluxo de eventos normal. Em muitos sistemas é necessário dar ao usuário a possibilidade de cancelar ou alterar dados de uma transação efetuada anteriormente com sucesso. Se cada uma dessas possibilidades for considerada como um caso de uso isolado, o número de casos de uso pode crescer demasiadamente, aumentando desnecessariamente a complexidade do modelo de casos de uso. Além disso, o fluxo de eventos normal de um caso de uso desse tipo tende a ser muito simples, não justificando documentar todo um conjunto de informações para adicionar apenas duas ou três linhas descrevendo os passos do caso de uso. Assim, em situações dessa natureza, é interessante considerar apenas um caso de uso, contendo diversos fluxos de eventos principais. Essa abordagem é bastante recomendada para casos de uso cadastrais, em que um único caso de uso inclui fluxos de eventos normais para criar, alterar, consultar e excluir entidades.

Fluxos de eventos normais podem ser descritos de diferentes maneiras, dependendo do nível de formalidade que se deseja para as descrições. Dentre os formatos possíveis, há dois principais:

- *Livre:* o fluxo de eventos normal é escrito na forma de um texto corrido, como no exemplo da Figura 5.11.
- *Enumerado:* cada passo do fluxo de eventos normal é numerado, de modo que possa ser referenciado nos fluxos de eventos alternativos ou em outros pontos do fluxo de eventos normal. A Figura 5.12 reapresenta o exemplo da Figura 5.11 neste formato. As seções iniciais foram omitidas por serem iguais às da Figura 5.11. Neste texto, advogamos em favor do uso do formato enumerado.

Cada exceção deve ser tratada por um *fluxo alternativo de exceção*. Fluxos alternativos de exceção devem ser descritos contendo as seguintes informações (WAZLAWICK, 2004): um identificador, uma descrição sucinta da exceção que ocorreu, os passos para tratar a exceção (ações corretivas) e uma indicação de como o caso de uso retorna ao fluxo principal (se for o caso) após a execução das ações corretivas.

Quando um formato de descrição enumerado é utilizado, não é necessário colocar uma verificação como uma condicional no fluxo principal. Por exemplo, no caso da Figura 5.12, o passo 3 não deve ser escrito como "3. Se o cartão é válido, o caixa automático solicita que o cliente informe a senha.". Basta o fluxo alternativo, no exemplo, o fluxo 2a.

Ainda quando o formato de descrição enumerado é utilizado, o identificador da exceção deve conter a linha do fluxo de eventos principal (ou eventualmente de algum outro fluxo de eventos alternativo) no qual a exceção ocorreu e uma letra para identificar a própria exceção (WAZLAWICK, 2004), como ilustra o exemplo da Figura 5.12.

Uma informação que precisa estar presente na descrição de um fluxo de eventos de exceção diz respeito a como finalizar o tratamento de uma exceção. Wazlawick (2004) aponta quatro formas básicas para finalizar o tratamento de uma exceção:

- Voltar ao início do caso de uso, o que não é muito comum nem prático.
- Voltar ao início do passo em que ocorreu a exceção e executá-lo novamente. Esta é a situação mais comum.
- Voltar para algum um passo posterior. Esta situação ocorre quando as ações corretivas realizam o trabalho que o passo (ou a sequência de passos) posterior deveria executar. Neste caso, é importante verificar se novas exceções não poderiam ocorrer.
- Abortar o caso de uso. Neste caso, não se retorna ao fluxo principal e o caso de uso não atinge seus objetivos.

Nome: Efetuar Saque

Fluxo de Eventos Normal

1. O cliente insere seu cartão no caixa automático.
2. O caixa automático analisa o cartão e verifica se ele é aceitável.
3. O caixa automático solicita que o cliente informe a senha.
4. O cliente informa a senha.
5. O caixa automático envia os dados do cartão e da senha para o sistema bancário para validação.
6. O caixa automático solicita que o cliente informe o tipo de transação a ser efetuada.
7. O cliente seleciona a opção saque.
8. O caixa automático solicita que seja informada a quantia.
9. O cliente informa a quantia a ser sacada.
10. O caixa automático envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada.
11. As notas são preparadas e liberadas.

Fluxos de Eventos de Exceção

2a – O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada e se retorna ao passo 1.

5a – Senha incorreta:
    5a.1 – 1ª e 2ª tentativas: Uma mensagem de erro é mostrada para o cliente. Retornar ao passo 3.
    5a.2 – 3ª tentativa: bloquear o cartão e abortar a transação.

10a - Saque não autorizado: Uma mensagem de erro é exibida e a operação é abortada.

11a - Não há dinheiro suficiente disponível no caixa eletrônico: Uma mensagem de erro é exibida e a operação é abortada.

1 a 9: Cancelamento: O cliente pode cancelar a transação, enquanto o saque não for autorizado pelo sistema bancário. A transação é abortada.

Figura 5.12 – Descrição do Caso de Uso *Efetuar Saque* – Formato Enumerado.

Além dos fluxos de exceção, há outro tipo de fluxo de eventos alternativo: os *fluxos variantes*. Fluxos variantes são considerados dentro da normalidade do caso de

uso e indicam formas diferentes, mas igualmente normais, de se realizar uma certa porção de um caso de uso. Seja o caso de um sistema de um supermercado, mais especificamente um caso de uso para efetuar uma compra. Um passo importante desse caso de uso é a realização do pagamento, o qual pode se dar de três maneiras distintas: pagamento em dinheiro, pagamento em cheque, pagamento em cartão. Nenhuma dessas formas de pagamento constitui uma exceção. São todas maneiras diferentes, mas normais, de realizar um certo passo do caso de uso e, portanto, pode-se dizer que o fluxo principal possui três variações. A descrição de um fluxo variante deve conter: um identificador, uma descrição sucinta do passo especializado e os passos enumerados, como ilustra a Figura 5.13.

Nome: Efetuar Compra

Fluxo de Eventos Normal

...
1. De posse do valor a ser pago, o atendente informa a forma de pagamento.
2. Efetuar o pagamento:
   2a. Em dinheiro
   2b. Em cheque
   2c. Em cartão
3. O pagamento é registrado.

Fluxos de Eventos Variantes

2a – Pagamento em Dinheiro:
- 2a.1 – O atendente informa a quantia em dinheiro entregue pelo cliente.
- 2a.2 – O sistema informa o valor do troco a ser dado ao cliente.

2b – Pagamento em Cheque:
- 2b.1 – O atendente informa os dados do cheque, a saber: banco, agência, conta e valor.

2c – Pagamento em Cartão:
- 2c.1 – O atendente informa os dados do cartão e o valor da compra.
- 2.c.2 – O sistema envia os dados informados no passo anterior, junto com a identificação da loja para o serviço de autorização do Sistema de Operadoras de Cartão de Crédito.
- 2c.3 – O Sistema de Operadoras de Cartão de Crédito autoriza a compra e envia o código da autorização.

Figura 5.13 – Descrição Parcial do Caso de Uso *Efetuar Compra* – com Variantes.

Por fim, em diversas situações, pode ser desnecessariamente trabalhoso especificar casos de uso segundo um formato completo, seja usando uma descrição dos fluxos de eventos no formato livre seja no formato enumerado. Para esses casos, um formato simplificado, na forma de uma tabela, pode ser usado. O formato tabular é normalmente empregado para casos de uso que possuem uma estrutura de interação simples, seguindo uma mesma estrutura geral, tais como casos de uso cadastrais (ou

CRUD[9]) e consultas. Casos de uso cadastrais de baixa complexidade tipicamente envolvem inclusão, alteração, consulta e exclusão de entidades e seguem o padrão de descrição mostrado na Figura 5.14.

Fluxos de Eventos Normais

Criar [Novo Objeto]
O [ator] informa os dados do [novo objeto], a saber: [atributos e associações do objeto]. Caso os dados sejam válidos, as informações são registradas.

Alterar Dados
O [ator] informa o [objeto] do qual deseja alterar dados e os novos dados. Os novos dados são validados e a alteração registrada.

Consultar Dados
O [ator] informa o [objeto] que deseja consultar. Os dados do [objeto] são apresentados.

Excluir [Objeto]
O [ator] informa o [objeto] que deseja excluir. Os dados do [objeto] são apresentados e é solicitada uma confirmação. Se a exclusão for confirmada, o [objeto] é excluído.

Fluxos de Eventos de Exceção

Incluir [Novo Objeto] / Alterar Dados
- Dados do [objeto] inválidos: uma mensagem de erro é exibida, solicitando correção da informação inválida.

Figura 5.14 – Padrão Típico de Descrição de Casos de Uso Cadastrais.

Assim, para simplificar a descrição de casos de uso cadastrais, recomenda-se utilizar o modelo tabular mostrado na Tabela 5.2. Quando essa tabela for empregada, estar-se-á assumindo que o caso de uso envolve os fluxos de eventos indicados (I para inclusão, A para alteração, C para consulta e E para exclusão), com a descrição base mostrada na Figura 5.14.

Tabela 5.2 – Modelo de Descrição de Casos de Uso Cadastrais

Caso de Uso	Ações Possíveis	Observações	Requisitos	Classes
<nome do caso de uso>	< I, A, C, E >			

A coluna **Observações** é usada para listar informações importantes relacionadas às ações, tais como os itens informados na inclusão, uma restrição a ser considerada para que a exclusão possa ser feita, uma informação que não pode ser alterada ou uma informação do objeto que não é apresentada na consulta. Deve-se indicar antes da

[9] CRUD – do inglês: Create, Read, Update and Delete; em português: Criar, Consultar, Atualizar e Excluir, ou seja, casos de uso que proveem as funções básicas de manipulação de dados de uma entidade de interesse do sistema.

observação a qual ação ela se refere ([I] para inclusão, [A] para alteração, [C] para consulta e [E] para exclusão).

As colunas **Requisitos** e **Classes** indicam, respectivamente, os requisitos que estão sendo (ou que devem ser) tratados pelo caso de uso e as classes do domínio do problema necessárias para a realização do caso de uso. O objetivo dessas colunas é manter a rastreabilidade dos casos de uso para requisitos e classes, respectivamente, de maneira análoga ao recomendado no formato completo.

A Tabela 5.3 ilustra a descrição de casos de usos cadastrais do subsistema Controle de Acervo de uma videolocadora.

Tabela 5.3 – Descrição de Casos de Uso Cadastrais – Controle de Acervo de Videolocadora

Caso de Uso	Ações Possíveis	Observações	Requisitos	Classes
Cadastrar Filme	I, A, C, E	[I] Informar: título original, título em português, país, ano, diretores, atores, sinopse, duração, gênero, distribuidora, tipo de áudio (p.ex., Dolby Digital 2.0), idioma do áudio e idioma da legenda. [E] Não é permitida a exclusão de filmes que tenham itens associados. [E] Ao excluir um filme, devem-se excluir as reservas associadas.	RF9, RNF1	Filme, Distribuidora
Cadastrar Item	I, A, C, E	[I] Informar: filme, tipo de mídia, data de aquisição e número de série. [E] Não é permitido excluir um item que tenha locações associadas.	RF9, RNF1, RNF3	Item, Filme, TipoMidia
Cadastrar Distribuidora	I, A, C, E	[I] Informar: razão social, CNPJ, endereço, telefone e pessoa de contato. [E] Não é permitido excluir uma distribuidora que tenha filmes associados.	RF10, RNF1	Distribuidora
Cadastrar Tipo de Mídia	I, A, C, E	[I] Informar: nome e valor de locação. [E] Não é permitido excluir um tipo de mídia que tenha itens associados. [E] Ao excluir um tipo de mídia, devem-se excluir as reservas que especificam apenas esse tipo de mídia.	RF9, RNF1	TipoMidia

Para casos de uso de consulta mais abrangente do que a consulta de um único objeto (já tratada como parte dos casos de uso cadastrais), mas ainda de baixa complexidade (tais como consultas que combinam informações de vários objetos envolvendo filtros), sugere-se utilizar o formato tabular mostrado na Tabela 5.4.

Tabela 5.4 – Modelo de Descrição de Casos de Uso de Consulta

Caso de Uso	Observações	Requisitos	Classes
<nome do caso de uso>			

A coluna **Observações** deve ser usada para listar informações importantes relacionadas à consulta, tais como dados que podem ser informados para a pesquisa, totalizações feitas em relatórios etc.

As colunas **Requisitos** e **Classes** têm a mesma função de suas homônimas no modelo da Tabela 5.2, ou seja, indicam, respectivamente, os requisitos que estão sendo tratados (ou que devem ser) pelo caso de uso e as classes do domínio do problema necessárias para a realização do mesmo.

A Tabela 5.5 ilustra a descrição de um caso de usos de consulta do subsistema Controle de Acervo de uma videolocadora.

Tabela 5.5 – Descrição de Casos de Uso de Consulta – Controle de Acervo de Videolocadora

Caso de Uso	Observações	Requisitos	Classes
Consultar Acervo	As consultas ao acervo poderão ser feitas informando uma (ou uma combinação) das seguintes informações: título (ou parte dele), original ou em português, gênero, tipo de mídia disponível, ator, diretor, nacionalidade e lançamentos.	RF11, RNF1, RNF2	Filme, Item, TipoMidia, Distribuidora

5.4.6 – Descrevendo Informações Complementares

As descrições dos fluxos de eventos principal, variantes e de exceção são cruciais em uma descrição de casos de uso. Contudo, há outras informações complementares que são bastante úteis e, portanto, que devem ser levantadas e documentadas como, por exemplo: pré-condições, pós-condições, requisitos relacionados e classes relacionadas.

Pré-condições estabelecem o que precisa ser verdadeiro antes de se iniciar um caso de uso. Pós-condições, por sua vez, estabelecem o que será verdadeiro após a execução do caso de uso. Pré-condições precisam ser verdadeiras para que o caso de uso possa ser iniciado. Não se deve confundi-las com exceções. Pré-condições não são testadas durante a execução do caso de uso (como ocorre com as condições que geram exceções). Ao contrário, elas são testadas antes de iniciar o caso de uso. Se a pré-condição é falsa, então não é possível executar o caso de uso. Para documentar as pré-condições, recomenda-se listar as condições que têm de ser satisfeitas na seção “Pré-condições”. Pré-condições devem ser escritas como uma simples asserção sobre o estado do mundo no momento em que o caso de uso inicia (COCKBURN, 2005).

Muitas vezes, uma pré-condição para ser atendida requer que um outro caso de uso já executado tenha estabelecido essa pré-condição. Contudo, um erro bastante comum é escrever como uma pré-condição algo que frequentemente, mas não necessariamente, é verdadeiro (COCKBURN, 2005). Seja o caso de uma locadora de vídeos em que clientes em atraso não podem locar novos itens até que regularize suas pendências. Neste caso, uma pré-condição do tipo “cliente não está em atraso” como pré-condição de um caso de uso “efetuar locação” é inadequada. Observe que a identificação do cliente é parte do caso de uso efetuar locação e, portanto, não é possível garantir que o cliente não está em atraso antes de iniciar o caso de uso. Esta situação tem de ser tratada como uma exceção e não como uma pré-condição.

As seções de requisitos e classes relacionados são importantes para a gerência de requisitos. A primeira estabelece um rastro entre casos de uso e os requisitos de usuário documentados no Documento de Requisitos, permitindo, em um primeiro momento, analisar se algum requisito não foi tratado. Em um segundo momento, quando uma alteração em um requisito é solicitada, é possível usar essa informação para analisar o impacto da alteração. Para documentar os requisitos relacionados, recomenda-se listar os identificadores de cada um dos requisitos na seção de "Requisitos Relacionados".

A seção de classes relacionadas indica quais são as classes do modelo conceitual estrutural necessárias para a realização do caso de uso. Essa seção permite rastrear casos de uso para classes em vários níveis, uma vez que há uma grande tendência de as mesmas classes do modelo conceitual estrutural estarem presentes nos modelos de projeto e no código fonte. Para documentar as classes relacionadas, recomenda-se listar o nome de cada uma das classes envolvidas na seção de "Classes Relacionadas". Vale ressaltar que essa informação é tipicamente preenchida durante a modelagem conceitual estrutural ou até mesmo depois, durante a elaboração de modelos de interação. A partir das informações de requisitos e classes relacionados, pode-se, por exemplo, construir matrizes de rastreabilidade.

5.4.7 - Relacionamentos entre Casos de Uso

Para permitir uma modelagem mais apurada dos casos de uso em um diagrama, três tipos de relacionamentos entre casos de uso podem ser empregados. Casos de uso podem ser descritos como versões especializadas de outros casos de uso (relacionamento de **generalização/ especialização**); casos de uso podem ser incluídos como parte de outro caso de uso (relacionamento de **inclusão**); ou casos de uso podem estender o comportamento de um outro caso de uso (relacionamento de **extensão**). O objetivo desses relacionamentos é tornar um modelo mais compreensível, evitar redundâncias entre casos de uso e permitir descrever casos de uso em camadas. A seguir esses tipos de relacionamentos são abordados.

Inclusão

Uma associação de inclusão de um *caso de uso base* para um *caso de uso de inclusão* significa que o comportamento definido no caso de uso de inclusão é incorporado ao comportamento do caso de uso base. Ou seja, a relação de inclusão incorpora um caso de uso (o caso de uso incluído) dentro da sequência de comportamento de outro caso de uso (o caso de uso base) (BLAHA; RUMBAUGH, 2006; OLIVÉ, 2007).

Esse tipo de associação é útil para extrair comportamento comum a vários casos de uso em uma única descrição, de modo que esse comportamento não tenha de ser descrito repetidamente. O caso de uso de inclusão pode ou não ser passível de utilização isoladamente. Assim, ele pode ser apenas um fragmento de uma funcionalidade, não precisando ser uma transação completa. A parte comum é incluída por todos os casos de uso base que têm esse caso de uso de inclusão em comum e a execução do caso de uso de inclusão é análoga a uma chamada de subrotina (OLIVÉ, 2007).

Na UML, o relacionamento de inclusão entre casos de uso é mostrado como uma dependência (seta pontilhada) estereotipada com a palavra-chave *include*, partindo do caso de uso base para o caso de uso de inclusão, como ilustra a Figura 5.15.

Figura 5.15 – Associação de Inclusão na UML

Uma associação de inclusão deve ser referenciada também na descrição do caso de uso base. O local em que esse comportamento é incluído deve ser indicado na descrição do caso de uso base, através de uma referência explícita à chamada ao caso de uso incluído. Assim, a descrição do fluxo de eventos (principal ou alternativo) do caso de uso base deve conter um passo que envolva a chamada ao caso de uso incluído, referenciada por "Incluir *nome do caso de uso incluído*". Para destacar referências de um caso de uso para outro, sugere-se que o nome do caso de uso referenciado seja sublinhado e escrito em itálico.

No exemplo do caixa automático, todos os três casos de uso têm em comum uma porção que diz respeito à validação inicial do cartão. Neste caso, um relacionamento de inclusão deve ser empregado, conforme mostra a Figura 5.16.
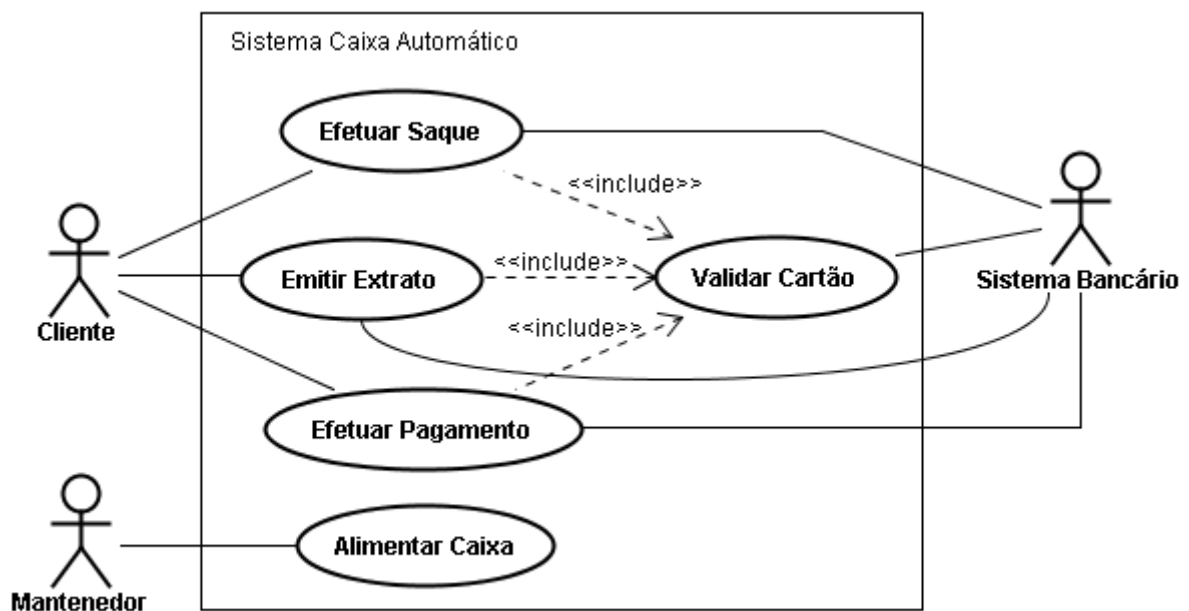
Figura 5.16 - Diagrama de Casos de Uso – Caixa Automático com Inclusão.

O caso de uso *Validar Cartão* extrai o comportamento descrito na Figura 5.17. Ao isolar este comportamento no caso de uso de *Validar Cliente*, o caso de uso *Efetuar Saque* passaria a apresentar a descrição mostrada na Figura 5.18.

Deve-se observar que não necessariamente o comportamento do caso de uso incluído precisa ser executado todas as vezes que o caso de uso base é realizado. Assim, é possível que a inclusão esteja associada a alguma condição. O caso de uso incluído é inserido em um local específico dentro da sequência do caso de uso base, da mesma forma que uma subrotina é chamada de um local específico dentro de outra subrotina (BLAHA; RUMBAUGH, 2006).

Nome: Validar Cartão

Fluxo de Eventos Normal

1. O cliente insere o cartão no caixa automático.
2. O caixa automático analisa o cartão e verifica se ele é aceitável.
3. O caixa automático solicita que o cliente informe a senha.
4. O cliente informa a senha.
5. O caixa automático envia os dados do cartão e da senha para o sistema bancário para validação.
6. O caixa automático solicita que o cliente informe o tipo de transação a ser efetuada.

Fluxos de Eventos de Exceção

2a – O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada e se retorna ao passo 1.

5a – Senha incorreta:

5a.1 – 1ª e 2ª tentativas: Uma mensagem de erro é mostrada para o cliente. Retornar ao passo 3.

5a.2 – 3ª tentativa: bloquear o cartão e abortar a transação.

1 a 5: Cancelamento: O cliente solicita o cancelamento da transação e a transação é abortada.

Figura 5.17 – Descrição do Caso de Uso *Validar Cartão*

Nome: Efetuar Saque

Fluxo de Eventos Normal

1. Incluir *Validar Cartão*.
2. O cliente seleciona a opção saque.
3. O caixa automático solicita que seja informada a quantia.
4. O cliente informa a quantia a ser sacada.
5. O caixa automático envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada.
6. As notas são preparadas e liberadas.

Fluxos de Eventos de Exceção

5a - Saque não autorizado: Uma mensagem de erro é exibida e a operação é abortada.

6a - Não há dinheiro suficiente disponível no caixa eletrônico: Uma mensagem de erro é exibida e a operação é abortada.

1 a 3: Cancelamento: O cliente pode cancelar a transação, enquanto o saque não for autorizado pelo sistema bancário. A transação é abortada.

Figura 5.18 – Descrição do Caso de Uso *Efetuar Saque* com inclusão.

Por fim, é importante frisar que não há um consenso sobre a possibilidade (ou não) de um caso de uso incluído poder ser utilizado isoladamente. Diversos autores, dentre eles Olivé (2007) e Blaha e Rumbaugh (2006), admitem essa possibilidade;

outros não. Em (BOOCH; RUMBAUGH; JACOBSON, 2006), diz-se explicitamente que um “caso de uso incluído nunca permanece isolado, mas é apenas instanciado como parte de alguma base maior que o inclui”. Neste texto, admitimos a possibilidade de um caso de uso incluído poder ser utilizado isoladamente, pois isso permite representar situações em que um caso de uso chama outro caso de uso (como uma chamada de subrotina), mas este último pode também ser realizado isoladamente. A Figura 5.19 ilustra uma situação bastante comum, em que, ao se realizar um processo de negócio (no caso a reserva de um carro em um sistema de locação de automóveis), caso uma informação necessária para esse processo (no caso o cliente) não esteja disponível, ela pode ser inserida no sistema. Contudo, o cadastro da informação também pode ser feito dissociado do processo de negócio que o inclui (no caso, o cliente pode se cadastrar fora do contexto da reserva de um carro). Ao não se admitir a possibilidade de um caso de uso incluído poder ser utilizado isoladamente, não é possível modelar situações desta natureza, as quais são bastante frequentes.
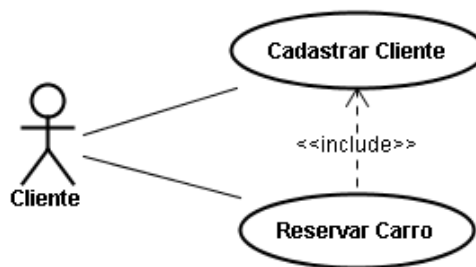
Figura 5.19 – Exemplo de Associação de Inclusão.

Extensão

Uma associação de extensão entre um *caso de uso de extensão* e um *caso de uso base* significa que o comportamento definido no caso de uso de extensão pode ser inserido dentro do comportamento definido no caso de uso base, em um local especificado indiretamente pelo caso de uso de extensão. A extensão ocorre em um ou mais pontos de extensão específicos definidos no caso de uso base. A extensão pode ser condicional. Neste caso, a extensão ocorre apenas se a condição é verdadeira quando o ponto de extensão especificado é atingido. O caso de uso base é definido de forma independente do caso de uso de extensão e é significativo independentemente do caso de uso de extensão (OLIVÉ, 2007; BOOCH; RUMBAUGH; JACOBSON, 2006).

Um caso de uso pode ter vários pontos de extensão e esses pontos são referenciados por seus nomes (BOOCH; RUMBAUGH; JACOBSON, 2006). O caso de uso base apenas indica seus pontos de extensão. O caso de uso de extensão especifica em qual ponto de extensão ele será inserido. Por isso, diz-se que o caso de uso de extensão especifica indiretamente o local onde seu comportamento será inserido.

A associação de extensão é como uma relação de inclusão olhada da direção oposta, em que a extensão se incorpora ao caso de uso base, em vez de o caso de uso base incorporar explicitamente a extensão. Ela conecta um caso de uso de extensão a um caso de uso base. O caso de uso de extensão é geralmente um fragmento, ou seja, ele não aparece sozinho como uma sequência de comportamentos. Além disso, na maioria das vezes, a relação de extensão possui uma condição associada e, neste caso, o comportamento de extensão ocorre apenas se a condição for verdadeira. O caso de uso

base, por sua vez, precisa ser, obrigatoriamente, um caso de uso válido na ausência de quaisquer extensões (BLAHA; RUMBAUGH, 2006).

Na UML, a associação de extensão entre casos de uso é mostrada como uma dependência (seta pontilhada) estereotipada com a palavra chave *extend*, partindo do caso de uso de extensão para o caso de uso base, como ilustra a Figura 5.20. Pontos de extensão podem ser indicados no compartimento da elipse do caso de uso, denominado "*extension points*" (pontos de extensão). Opcionalmente, a condição a ser satisfeita e a referência ao ponto de extensão podem ser mostradas por meio de uma nota[10] anexada à associação de extensão (OLIVÉ, 2007). Assim, no exemplo da Figura 5.20, o *Caso de Uso de Extensão 1* é executado quando o *ponto de extensão 1* do *Caso de Uso Base* for atingido, se a *condição* for verdadeira.
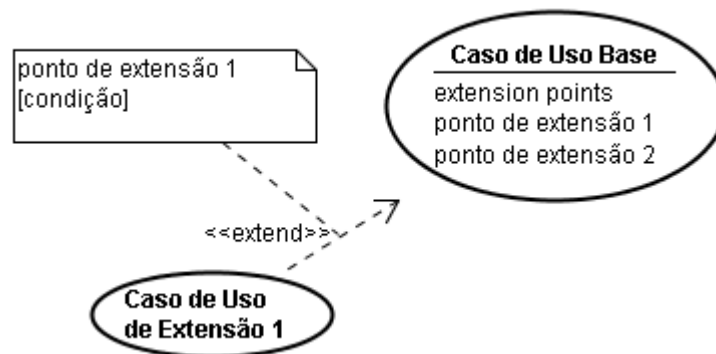
Figura 5.20 – Associação de Extensão na UML

Uma importante diferença entre as associações de inclusão e extensão é que, na primeira o caso de uso base está ciente do caso de uso de inclusão, enquanto na segunda o caso de uso base não está ciente dos possíveis casos de uso de extensão (OLIVÉ, 2007).

Assim como no caso da inclusão, uma associação de extensão deve ser referenciada na descrição do caso de uso base. Neste caso, contudo, o caso de uso base apenas aponta o ponto de extensão, sem fazer uma referência explícita ao caso de uso de extensão. O local de cada um dos pontos de extensão deve ser indicado na descrição do caso de uso base, através de uma referência ao nome do ponto de extensão seguido de ": ponto de extensão". Assim, a descrição do fluxo de eventos (principal ou alternativo) do caso de uso base deve conter indicações explícitas para cada ponto de extensão.

No exemplo do caixa automático, suponha que se deseja coletar dados estatísticos sobre os valores das notas entregues nos saques, de modo a permitir alimentar o caixa eletrônico com as notas mais adequadas para saque. Poder-se-ia, então, estender o caso de uso *Efetuar Saque*, de modo que, quando necessário, outro caso de uso, denominado *Coletar Estatísticas de Notas*, contasse e acumulasse o tipo das notas entregues em um saque, conforme mostra a Figura 5.21. A Figura 5.22 mostra

[10] Nota é o único item de anotação da UML. Notas são usadas para explicar partes de um modelo da UML. São comentários incluídos para descrever, esclarecer ou fazer alguma observação sobre qualquer elemento do modelo. Assim, uma nota é apenas um símbolo para representar restrições e comentários anexados a um elemento ou a uma coleção de elementos. Graficamente, uma nota é representada por um retângulo com um dos cantos com uma dobra de página, acompanhado por texto e anexada ao(s) elemento(s) anotados por meio de linha(s) pontilhada(s) (BOOCH; RUMBAUGH; JACOBSON, 2006). No exemplo da Figura 5.20, a nota está anexada ao relacionamento de extensão, adicionando-lhe informações sobre o ponto de extensão e a condição associados à extensão.

a descrição do caso de uso *Efetuar Saque* indicando o ponto de extensão *entrega do dinheiro*.
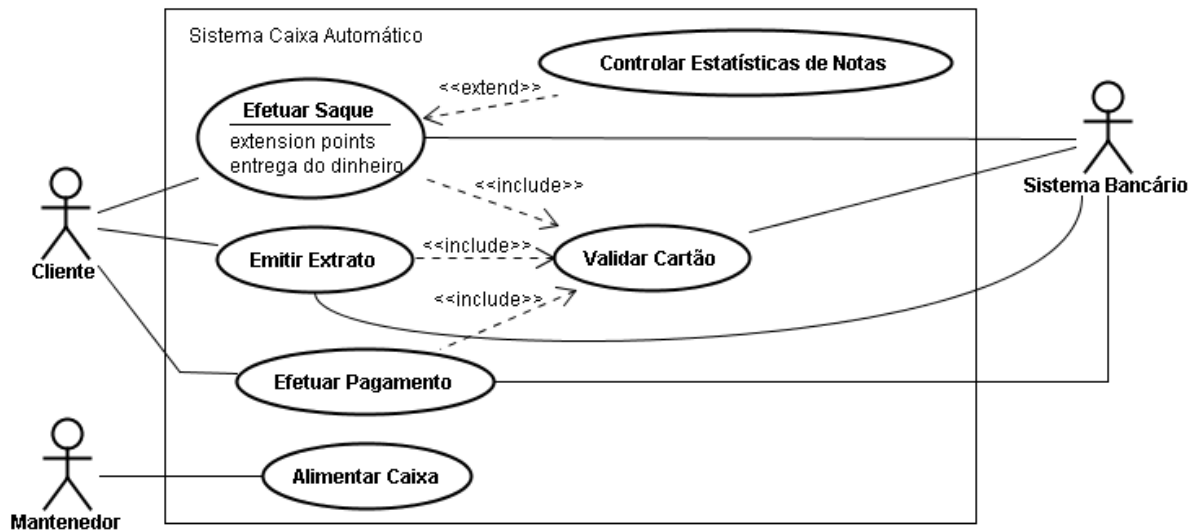
Figura 5.21 - Diagrama de Casos de Uso – Caixa Automático com Extensão.

Nome: Efetuar Saque

Fluxo de Eventos Normal

1. Incluir <u>*Validar Cartão*</u>.
2. O cliente seleciona a opção saque.
3. O caixa automático solicita que seja informada a quantia.
4. O cliente informa a quantia a ser sacada.
5. O caixa automático envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada.
6. As notas são preparadas.

entrega do dinheiro: ponto de extensão.

7. As notas são liberadas

Fluxos de Eventos de Exceção

5a - Saque não autorizado: Uma mensagem de erro é exibida e a operação é abortada.

6a - Não há dinheiro suficiente disponível no caixa eletrônico: Uma mensagem de erro é exibida e a operação é abortada.

1 a 3: Cancelamento: O cliente pode cancelar a transação, enquanto o saque não for autorizado pelo sistema bancário. A transação é abortada.

Figura 5.22 – Descrição do Caso de Uso *Efetuar Saque* com inclusão.

Generalização / Especialização

Um relacionamento de generalização / especialização entre um *caso de uso pai* e um *caso de uso filho* significa que o caso de uso filho herda o comportamento e o significado do caso de uso pai, acrescentando ou sobrescrevendo seu comportamento (OLIVÉ, 2007; BOOCH; RUMBAUGH; JACOBSON, 2006). Na UML, relacionamentos de generalização / especialização são representados como uma linha cheia direcionada com uma seta aberta (símbolo de herança), como ilustra a Figura 5.22.

Figura 5.22 – Associação de Generalização / Especialização entre Casos de Uso na UML.

Voltando ao exemplo do sistema de caixa automático, suponha que haja duas formas adotadas para se validar o cartão: a primeira através de senha, como descrito anteriormente, e a segunda por meio de análise da retina do cliente. Neste caso, poderiam ser criadas duas especializações do caso de uso *Validar Cliente*, como mostra a Figura 5.23.
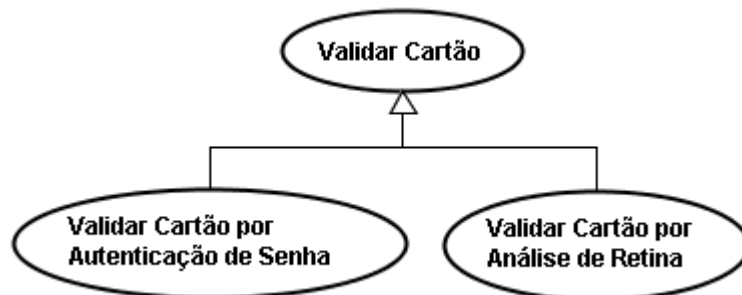
Figura 5.23 – Exemplo de Generalização / Especialização entre Casos de Uso

A descrição do caso de uso pai teria de ser generalizada para acomodar diferentes tipos de validação. Esses tipos de validação seriam especializados nas descrições dos casos de uso filhos. A Figura 5.24 mostra as descrições desses três casos de uso.

A generalização / especialização é aplicável quando um caso de uso possui diversas variações. O comportamento comum pode ser modelado como um caso de uso abstrato e especializado para as diferentes variações (BLAHA; RUMBAUGH, 2006). Contudo, avalie se não fica mais simples e direto descrever essas variações como fluxos alternativos variantes na descrição de casos de uso. Quando forem poucas e pequenas as variações, muito provavelmente será mais fácil capturá-las na descrição, ao invés de criar hierarquias de casos de uso. A Figura 5.25 mostra uma solução análoga à da Figura 5.24, sem usar, no entanto, especializações do caso de uso.

Nome: Validar Cartão

Fluxo de Eventos Normal

1. O cliente insere o cartão no caixa automático.
2. O caixa automático analisa o cartão e verifica se ele é aceitável.
3. O caixa automático solicita informação para identificação do cliente.
4. O cliente informa sua identificação.
5. O caixa automático envia os dados do cartão e da identificação para o sistema bancário para validação.
6. O caixa automático solicita que o cliente informe o tipo de transação a ser efetuada.

Fluxos de Eventos de Exceção

2a – O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada e se retorna ao passo 1.

5a – Dados de Identificação Incorretos:

5a.1 – 1ª e 2ª tentativas: Uma mensagem de erro é mostrada para o cliente. Retornar ao passo 3.

5a.2 – 3ª tentativa: bloquear o cartão e abortar a transação.

1 a 5: Cancelamento: O cliente solicita o cancelamento da transação e a transação é abortada.

Nome: Validar Cartão por Análise de Retina

Fluxo de Eventos Normal

3. O caixa automático solicita que o cliente se posicione corretamente para a captura da imagem da retina.
4. O caixa automático retira uma foto da retina do cliente.
5. O caixa automático envia os dados do cartão e a foto da retina para o sistema bancário para validação.

Nome: Validar Cartão por Autenticação de Senha

Fluxo de Eventos Normal

3. O caixa automático solicita a senha.
4. O cliente informa a senha.
5. O caixa automático envia os dados do cartão e a senha para o sistema bancário para validação.

Figura 5.24 – Descrição do Caso de Uso *Validar Cartão* e suas Especializações.

Nome: Validar Cartão

Fluxo de Eventos Normal

1. O cliente insere o cartão no caixa automático.
2. O caixa automático analisa o cartão e verifica se ele é aceitável.
3. Validar cartão.
4. O caixa automático solicita que o cliente informe o tipo de transação a ser efetuada.

Fluxos de Eventos Variantes

3a – Validar cartão por autenticação de senha:

3a.1 – O caixa automático solicita a senha.

3a.2 – O cliente informa a senha.

3a.3 – O caixa automático envia os dados do cartão e a senha para o sistema bancário para validação.

3b – Validar cartão por análise de retina:

3b.1 – O caixa automático solicita que o cliente se posicione corretamente para a captura da imagem da retina.

3b.2 – O caixa automático retira uma foto da retina do cliente.

3b.3 – O caixa automático envia os dados do cartão e a foto da retina para o sistema bancário para validação.

Fluxos de Eventos de Exceção

2a – O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada e se retorna ao passo 1.

5a – Dados de Identificação Incorretos:

5a.1 – 1ª e 2ª tentativas: Uma mensagem de erro é mostrada para o cliente. Retornar ao passo 3.

5a.2 – 3ª tentativa: bloquear o cartão e abortar a transação.

1 a 5: Cancelamento: O cliente solicita o cancelamento da transação e a transação é abortada.

Figura 5.25 – Descrição do Caso de Uso *Validar Cartão* com Variantes.

Diretrizes para o Uso dos Tipos de Relacionamentos entre Casos de Uso

Os relacionamentos entre casos de uso devem ser utilizados com cuidado para evitar a introdução de complexidade desnecessária. As seguintes orientações são úteis para ajudar a decidir quando usar relacionamentos entre casos de uso em um diagrama de casos de uso:

- A inclusão é tipicamente aplicável quando se deseja capturar um fragmento de comportamento comum a vários casos de uso. Na maioria das vezes, o

caso de uso de inclusão é uma atividade significativa, mas não como um fim em si mesma (BLAHA; RUMBAUGH, 2006). Ou seja, o caso de uso de inclusão não precisa ser uma transação completa.

- Um relacionamento de inclusão é empregado quando há uma porção de comportamento que é similar ao longo de um ou mais casos de uso e não se deseja repetir a sua descrição. Para evitar redundância e assegurar reúso, extrai-se essa descrição e se compartilha a mesma entre diferentes casos de uso.Desta maneira, utiliza-se a inclusão para evitar ter de descrever o mesmo fragmento de comportamento várias vezes, capturando o comportamento comum em um caso de uso próprio (BOOCH; RUMBAUGH; JACOBSON, 2006).
- Não se deve utilizar o relacionamento de generalização / especialização para compartilhar fragmentos de comportamento. Para este propósito, deve-se usar a relação de inclusão (BLAHA; RUMBAUGH, 2006).
- A relação de extensão é bastante útil em situações em que se pode definir um caso de uso significativo com recursos adicionais. O comportamento básico é capturado no caso de uso base e os recursos adicionais nos casos de uso de extensão. Use a relação de extensão quando o sistema puder ser usado em diferentes configurações, algumas com os recursos adicionais e outras sem eles (BLAHA; RUMBAUGH, 2006).
- Tanto a inclusão quanto a extensão podem ser usadas para dividir o comportamento em partes menores. A inclusão, entretanto, implica que o comportamento incluído é uma parte necessária de um sistema configurado, mesmo que seu comportamento não seja executado todas as vezes, ou seja, mesmo que o comportamento incluído esteja associado a uma condição. A extensão, por sua vez, implica que o sistema sem o comportamento adicionado pela extensão é significativo (BLAHA; RUMBAUGH, 2006).

5.5 – Modelagem Conceitual Estrutural

O modelo conceitual estrutural de um sistema tem por objetivo descrever as informações que esse sistema deve representar e gerenciar.Modelos conceituais devem ser concebidos com foco no domínio do problema e não no domínio da solução e, por conseguinte, um modelo conceitual estrutural é um artefato do domínio do problema e não do domínio da solução.

As informações a serem capturadas em um modelo conceitual estrutural devem existir independentemente da existência de um sistema computacional para tratá-las. Ele deve ser independe da solução computacional a ser adotada para resolver o problema e deve conter apenas os elementos de informação referentes ao domínio do problema em questão. Elementos da solução, tais como interfaces, formas de armazenamento e comunicação, devem ser tratados apenas na fase de projeto (WAZLAWICK, 2004).

Uma vez que requisitos não-funcionais de produto (atributos de qualidade) são inerentes à solução computacional, geralmente eles não são tratados na modelagem conceitual. Ou seja, não se consideram elementos de informação para tratar aspectos como desempenho, segurança de acesso, confiabilidade, formas de armazenamento etc.

Esses atributos de qualidade do produto são considerados posteriormente, na fase de projeto.

Os elementos de informação básicos da modelagem conceitual estrutural são os tipos de entidades e os tipos de relacionamentos. A identificação de quais os tipos de entidades e os tipos de relacionamentos que são relevantes para um particular sistema de informação é uma meta crucial da modelagem conceitual (OLIVÉ, 2007).

Na modelagem conceitual segundo o paradigma orientado a objetos, tipos de entidades são modelados como classes. Tipos de relacionamentos são modelados como atributos e associações. Assim, o propósito da modelagem conceitual estrutural orientada a objetos é definir as classes, atributos e associações que são relevantes para tratar o problema a ser resolvido. Para tal, as seguintes tarefas devem ser realizadas:

- Identificação de Classes
- Identificação de Atributos e Associações
- Especificação de Hierarquias de Generalização/Especialização

É importante notar que essas atividades são dependentes umas das outras e que, durante o desenvolvimento, elas são realizadas, tipicamente, de forma paralela e iterativa, sempre visando ao entendimento do domínio do problema, desconsiderando aspectos de implementação.

5.5.1 Identificação de Classes

Classificação é o meio pelo qual os seres humanos estruturam a sua percepção do mundo e seu conhecimento sobre ele. Sem ela, não é possível nem entender o mundo a nossa volta nem agir sobre ele. Classificação assume a existência de tipos e de objetos a serem classificados nesses tipos. Classificar consiste, então, em determinar se um objeto é ou não uma instância de um tipo. A classificação nos permite estruturar conhecimento sobre as coisas em dois níveis: tipos e instâncias. No nível de tipos, procuramos encontrar as propriedades comuns a todas as instâncias de um tipo. No nível de instância, procuramos identificar o tipo do qual o objeto é uma instância e os valores particulares das propriedades desse objeto (OLIVÉ, 2007).

Tipos de entidade são um dos mais importantes elementos em modelos conceituais. Definir os tipos de entidade relevantes para um particular sistema de informação é uma tarefa crucial na modelagem conceitual. Um tipo de entidade pode ser definido como um tipo cujas instâncias em um dado momento são objetos individuais identificáveis que se consideram existir no domínio naquele momento. Um objeto pode ser instância de vários tipos ao mesmo tempo (OLIVÉ, 2007). Por exemplo, seja o caso dos tipos *Estudante* e *Funcionário* em um sistema de uma universidade. Uma mesma pessoa, por exemplo João, pode ser ao mesmo tempo um estudante e um funcionário dessa universidade.

Na orientação a objetos, tipos de entidade são representados por classes, enquanto as instâncias de um tipo de entidade são objetos. Assim, uma atividade crucial da modelagem conceitual estrutural segundo o paradigma orientado a objetos (OO) é a identificação de classes. Na UML, classes são representadas por um retângulo com três compartimentos: o compartimento superior é relativo ao nome da classe; o compartimento do meio é dedicado à especificação dos atributos da classe; e o

compartimento inferior é dedicado à especificação das operações da classe. A Figura 5.26 mostra a notação de classe na UML.
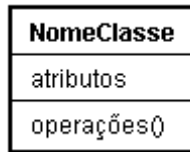
Figura 5.26 – Notação de Classes na UML.

Para nomear classes, sugere-se iniciar com um substantivo no singular, o qual pode ser combinado com complementos ou adjetivos, omitindo-se preposições. O nome da classe deve ser iniciado com letra maiúscula, bem como os nomes dos complementos, sem dar um espaço em relação à palavra anterior. Acentos não devem ser utilizados. Ex.: Cliente, PessoaFisica, ItemPedido.

Tomando por base os requisitos iniciais do usuário e, sobretudo, o modelo de casos de uso, é possível iniciar o trabalho de modelagem da estrutura do sistema. Esse trabalho começa com a descoberta de quais classes devem ser incluídas no modelo. O cerne de um modelo OO é exatamente o seu conjunto de classes.

Durante a análise de requisitos, tipicamente o analista estuda, filtra e modela o domínio do problema. Dizemos que o analista "filtra" o domínio, pois apenas uma parte desse domínio fará parte das responsabilidades do sistema. Assim, um domínio de problemas pode incluir várias informações, mas as responsabilidades de um sistema nesse domínio podem incluir apenas uma pequena parcela deste conjunto.

As classes de um modelo representam a expressão inicial do sistema. As atividades subsequentes da modelagem estrutural buscam obter uma descrição cada vez mais detalhada, em termos de associações e atributos. Contudo, deve-se observar que, à medida que atributos e associações vão sendo identificados, se ganha maior entendimento a respeito do domínio e naturalmente novas classes surgem. Assim, as atividades da modelagem conceitual são iterativas e com alto grau de paralelismo, devendo ser realizadas concomitantemente.

Conforme apontado anteriormente, dois importantes insumos para a atividade de identificação de classes são o Documento de Requisitos e o Modelo de Casos de Uso. Uma maneira bastante prática e eficaz de trabalhar a identificação de classes consiste em olhar esses dois documentos, em especial a descrição do minimundo e as descrições de casos de uso, à procura de classes.

Diversos autores, dentre eles Jacobson (1992) e Wazlawick (2004), sugerem que uma boa estratégia para identificar classes consiste em ler esses documentos procurando por substantivos. Esses autores argumentam que uma classe é, tipicamente, descrita por um nome no domínio e, portanto, aprender sobre a terminologia do domínio do problema é um bom ponto de partida. Ainda que um bom ponto de partida, essa heurística é ainda muito vaga. Se o analista segui-la fielmente, muito provavelmente ele terá uma extensa lista de potenciais classes, sendo que muitas delas podem, na verdade, se referir a atributos de outras classes. Além disso, pode ser que importantes classes não sejam capturadas, notadamente aquelas que se referem ao registro de eventos de negócio, uma vez que esses eventos muitas vezes são descritos na forma de verbos. Seja o seguinte exemplo de uma descrição de um domínio de locação de automóveis: "clientes locam carros". Seriam consideradas potenciais classes: *Cliente* e *Carro*.

Contudo, a locação é um evento de negócio importante que precisa ser registrado e, usando a estratégia de identificar classes a partir de substantivos, *Locação* não entraria na lista de potenciais classes.

Assim, neste texto sugere-se que, ao examinar documentos de requisitos e modelos de casos de uso, os seguintes elementos sejam considerados como candidatos a classes:

- Agentes: entidades do domínio do problema que têm a capacidade de agir com intenção de atingir uma meta. Em sistemas de informação, há dois tipos principais de agentes: os agentes físicos (tipicamente pessoas) e os agentes sociais (organizações, unidades organizacionais, sociedades etc.). Em relação às pessoas, deve-se olhar para os papéis desempenhados pelas diferentes pessoas no domínio do problema.
- Objetos: entidades sem a capacidade de agir, mas que fazem parte do domínio de informação do problema. Podem ser também classificados em físicos (p.ex., carros, livros, imóveis) e sociais (p.ex., cursos, disciplinas, leis). Entretanto, há também outros tipos de objetos, tais como objetos de caráter descrito usado para organizar e descrever outros objetos de um domínio (p.ex., modelos de carro), algumas vezes denominados objetos de especificação. Objetos sociais e de descrição (ou especificação) tendem a ser coisas menos tangíveis, mas são tão importantes para a modelagem conceitual quanto os objetos físicos e, portanto, palpáveis.
- Eventos: representam a ocorrência de ações no domínio do problema que precisam ser registradas e lembradas pelo sistema. Eventos acontecem no tempo e, portanto, a representação de eventos normalmente envolve a necessidade de registrar, dentre outros, quando o evento ocorreu. Deve-se observar que muitos eventos ocorrem no domínio do problema, mas grande parte deles não precisa ser lembrada. Para capturar os eventos que precisam ser lembrados e, portanto, registrados, devem-se focalizar os principais eventos de negócio do domínio do problema. Assim, em um sistema de locação de automóveis, são potenciais classes de eventos: *Locação*, *Devolução* e *Reserva*. Por outro lado, a ocorrência de eventos cadastrais, tais como os cadastros de clientes e carros, tende a ser de pouca importância, não sendo necessário lembrar a ocorrência desses eventos.

Seja qual for a estratégia usada para identificar classes, é sempre importante que o analista tenha em mente os objetivos do sistema durante a modelagem conceitual. Não se devem representar informações irrelevantes para o sistema e, portanto, a relevância para o sistema é o principal critério a ser adotado para decidir se um determinado elemento deve ou não ser incluído no modelo conceitual estrutural do sistema.

O resultado principal da atividade de identificação de classes é a obtenção de uma lista de potenciais classes para o sistema em estudo. Um modelo conceitual estrutural para uma aplicação complexa pode ter dezenas de classes e, portanto, pode ser necessário definir uma representação concisa capaz de orientar um leitor em um modelo desta natureza. O agrupamento de classes em subsistemas serve basicamente a este propósito, podendo ser útil também para a organização de grupos de trabalho em projetos extensos. Através da identificação e agrupamento de classes em subsistemas, é possível controlar a visibilidade do leitor e, assim, tornar o modelo mais compreensível.

Assim, da mesma maneira que casos de uso são agrupados em pacotes, classes também devem ser.

Quando uma coleção de classes colabora entre si para realizar um conjunto coeso de responsabilidades (casos de uso), elas podem ser vistas como um subsistema. Assim, um subsistema é uma abstração que provê uma referência para mais detalhes em um modelo de análise, incluindo tanto casos de uso quanto classes. O agrupamento de classes em subsistemas permite apresentar o modelo global em uma perspectiva mais alta. Esse nível ajuda o leitor a rever o modelo, bem como constitui um bom critério para organizar a documentação.

Uma vez identificadas as potenciais classes, deve-se proceder uma avaliação para decidir o que efetivamente considerar ou rejeitar. Conforme discutido anteriormente, a relevância para o sistema deve ser o critério principal. Além desse critério, os seguintes também devem ser considerados nessa avaliação:

- *Estrutura complexa:* o sistema precisa tratar informações sobre os objetos da classe? Tipicamente, uma classe deve ter, pelo menos, dois atributos. Se uma classe apresentar apenas um atributo, avalie se não é melhor tratá-la como um atributo de uma classe existente[11].
- *Atributos e associações comuns*: os atributos e as associações da classe devem ser aplicáveis a todas as suas instâncias, isto é, a todos os objetos da classe.
- *Classes não redundantes:* duas classes são redundantes quando elas têm sempre a mesma população[12]. Seja o exemplo de um modelo conceitual que tenha as classes Pessoa e Funcionário. Se o sistema está interessado apenas nas pessoas empregadas na organização (ou seja, funcionários), então a população dessas duas classes será sempre a mesma. A introdução de classes redundantes afeta e simplicidade do modelo e, portanto, um modelo conceitual não deve incluir classes redundantes.
- *Existência de instâncias:* toda classe deve possuir uma população não vazia. Uma potencial classe que possui uma única instância também não deve ser considerada uma classe. Tipicamente uma classe possui várias instâncias e a população da classe varia ao longo do tempo.

5.5.2 - Identificação de Atributos e Associações

Conforme apontado anteriormente, uma classe típica de um modelo conceitual estrutural deve apresentar estrutura complexa. A estrutura de uma classe corresponde a seus atributos e associações.

Conceitualmente, não há diferença entre atributos e associações. Atributos são, na verdade, tipos de relacionamentos binários. Em um tipo de relacionamento binário, há dois participantes. Em alguns tipos de relacionamentos, esses participantes são considerados "colegas", porque eles desempenham funções análogas e nenhum deles é subordinado ao outro. Seja o caso do tipo de relacionamento "*aluno cursa um curso*".

[11] Uma classe que possui um único atributo, mas várias associações, também satisfaz a esse critério.
[12] A população de uma classe em um dado momento é o conjunto de instâncias que existem no domínio naquele momento (OLIVÉ, 2007).

Um aluno não pode cursar sem haver um curso, bem como um curso não pode ser cursado se não houver um aluno. A ordem dos participantes no modelo não implica uma relação de prioridade ou subordinação entre eles (OLIVÉ, 2007). Na orientação a objetos, esse tipo de relacionamento é modelado como uma associação.

Entretanto, há alguns tipos de relacionamentos nos quais usuários e analistas consideram um participante como sendo uma característica do outro. Seja o exemplo do tipo de relacionamento "*filme possui gênero*". Alguém pode argumentar que o participante *gênero* é uma característica de filme e, portanto, subordinado a este. Esse tipo de relacionamento é modelado como um atributo. Assim, um atributo é um tipo de relacionamento binário em que um participante é considerado uma característica de outro. Por conseguinte, um atributo é igual a uma associação, exceto pelo fato de usuários e analistas adicionarem a interpretação que um dos participantes é subordinado ao outro (OLIVÉ, 2007).

De uma perspectiva mais prática, atributos podem ser vistos como informações alfanuméricas ligadas a um conceito. Associações, por sua vez, consistem em um tipo de informação que liga diferentes conceitos entre si (WAZLAWICK, 2004). Atributos ligam classes do domínio do problema a tipos de dados.

Tipos de dados podem ser primitivos ou específicos de domínio. Os tipos de dados primitivos são aplicáveis aos vários domínios e sistemas, tais como strings, datas, inteiros e reais, e são considerados como sendo predefinidos. Os tipos de dados específicos de um domínio de aplicação, por outro lado, precisam ser definidos. São exemplos de tipos de dados específicos: CPF, ISBN de livros, endereço etc.

Neste texto são considerados os seguintes tipos de dados primitivos:

- *String*: cadeia de caracteres;
- *boolean*: admite apenas os valores verdadeiro e falso;
- *Integer* (ou *int*): números inteiros;
- *Float* (ou *float*): números reais;
- *Currency*: valor em moeda (reais, dólares etc.);
- *Date*: datas, com informação de dia, mês e ano;
- *Time*: horas em um dia, com informação de hora, minuto e segundo;
- *DateTime*: combinação dos dois anteriores;
- *YearMonth*: informação de tempo contendo apenas mês e ano;
- *Year*: informação de tempo contendo apenas ano.

Atributos

Um atributo é uma informação de estado para a qual cada objeto em uma classe tem o seu próprio valor. Os atributos adicionam detalhes às abstrações e são apresentados na parte central do símbolo de classe.

Conforme discutido anteriormente, atributos possuem um tipo de dado, que pode ser primitivo ou específico de domínio. Ao identificar um atributo como sendo relevante, deve-se definir qual o seu tipo de dado. Caso nenhum dos tipos de dados

primitivos se aplique, deve-se definir, então, um tipo de dados específico. Por exemplo, em domínios que lidem com livros, é necessário definir o tipo ISBN[13], cujas instâncias são ISBNs válidos. Em domínios que lidem com pessoas físicas e jurídicas, CPF e CNPJ também devem ser definidos como tipos de dados específicos. Usar um tipo de dados primitivo nestes casos, tais como *String* ou *int*, é insuficiente, pois não são quaisquer cadeias de caracteres ou números que se caracterizam como ISBNs, CPFs ou CNPJs válidos.

Tipos de dados específicos podem apresentar propriedades. Por exemplo, CPF é um número de 11 dígitos, que pode ser dividido em duas partes: os 9 primeiros dígitos e os dois últimos, que são dígitos verificadores.

Um tipo de dados especial é a enumeração. Na enumeração, os valores do tipo são enumerados explicitamente na forma de literais, como é o caso do tipo *DiaSemana*, que é tipicamente definido como um tipo de dados compreendendo sete valores: Segunda, Terça, Quarta, Quinta, Sexta, Sábado e Domingo. É importante observar que tipos de dados enumerados só devem ser usados quando se sabe à priori quais são os seus valores e eles são fixos. Assim, são bons candidatos a tipos enumerados informações como sexo (M/F), estado civil etc.

Tipos de dados geralmente não são representados graficamente em um modelo conceitual estrutural, de modo a torná-lo mais simples. Na maioria das situações, basta descrever os tipos de dados específicos de domínio no Dicionário de Dados do Projeto. Contudo, se necessário, eles podem ser representados graficamente usando o símbolo de classe estereotipado com a palavra chave <<*dataType*>>. Tipos enumerados também podem ser representados usando o símbolo de classe, mas com o estereótipo <<*enumeration*>>, sendo que ao invés de apresentar atributos de um tipo de dados, enumeram-se os valores possíveis da enumeração. A Figura 5.27 ilustra a notação de tipos de dados na UML.
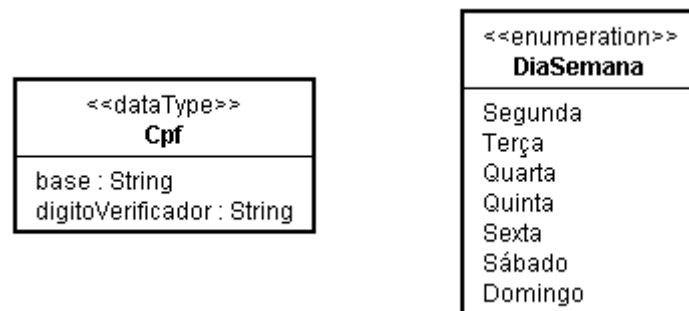
Figura 5.27 – Notação de Tipos de Dados na UML.

Uma dúvida típica e recorrente na modelagem estrutural é se um determinado item de informação deve ser modelado como uma classe ou como um atributo. Para que o item seja considerado uma classe, ele tem de passar nos critérios de inclusão no modelo discutidos na seção anterior. Entretanto, há alguns itens de informação que passam nesses critérios, mas que ainda assim podem ser melhor modelados como atributos, tendo como tipo um tipo de dado complexo, específico de domínio. Um

[13] O ISBN - *International Standard Book Number* - é um sistema internacional padronizado que identifica numericamente os livros segundo o título, o autor, o país, a editora, individualizando-os inclusive por edição. Utilizado também para identificar software, seu sistema numérico pode ser convertido em código de barras.

atributo deve capturar um conceito atômico, i.e., um único valor ou um agrupamento de valores fortemente relacionados que sirva para descrever um outro objeto. Além disso, para que um item de estrutura complexa seja modelado como um atributo, ele deve ser compreensível pelos interessados simplesmente pelo seu nome.

É bom realçar que, com o tempo, as classes do domínio do problema tendem a permanecer relativamente estáveis, enquanto os atributos provavelmente se alteram. Atributos podem ser bastante voláteis, em função de alterações nas responsabilidades do sistema.

É muito importante lembrar também que, uma vez que atributos e associações são tipos de relacionamentos, não devemos incluir na lista de atributos de uma classe, atributos representando associações (ou atributos representando “chaves estrangeiras” como a classe fosse uma tabela de um banco de dados relacional). Associações já têm sua presença indicada pela notação de associação, ou seja pelas linhas que conectam as classes que se relacionam.

Um aspecto bastante importante na especificação de atributos é a escolha de nomes. Deve-se procurar utilizar o vocabulário típico do domínio do problema, usando nomes legíveis e abrangentes. Para nomear atributos, sugerem-se nomes iniciando com substantivo, o qual pode ser combinado com complementos ou adjetivos, omitindo-se preposições. O nome do atributo deve ser iniciado com letra minúscula, enquanto os nomes dos complementos devem iniciar com letras maiúsculas, sem dar um espaço em relação à palavra anterior. Acentos não devem ser utilizados. Atributos monovalorados devem iniciar com substantivo no singular (p.ex., nome, razaoSocial), enquanto atributos multivalorados devem iniciar com o substantivo no plural (p.ex., telefones).

A sintaxe de atributos na UML, em sua forma plena, é a seguinte (BOOCH; RUMBAUGH; JACOBSON, 2006):

visibilidade nome: tipo [multiplicidade] = valorInicial {propriedades}

A visibilidade de um atributo indica em que situações esse atributo é visível por outras classes. Na UML há quatro níveis de visibilidade, os quais são marcados pelos seguintes símbolos:

+ público : o atributo pode ser acessado por qualquer classe;

protegido: o atributo só é passível de acesso pela própria classe ou por uma de suas especializações;

- privado: o atributo só pode ser acessado pela própria classe;

~ pacote: o atributo só pode ser acessado por classes declaradas dentro do mesmo pacote da classe a que pertence o atributo.

A informação de visibilidade é inerente à fase de projeto e não deve ser expressa em um modelo conceitual. Assim, em um modelo conceitual, atributos devem ser especificados sem nenhum símbolo antecedendo o nome.

O *tipo* indica o tipo de dado do atributo, o qual deve ser um tipo de dado primitivo ou um tipo de dado específico de domínio. Tipos de dados específicos de domínio devem ser definidos no Dicionário de Dados do Projeto. Para tornar os modelos conceituais mais simples, de modo a facilitar a comunicação com clientes e usuários, tipos de dados de atributos podem ser omitidos do diagrama de classes.

A multiplicidade é a especificação do intervalo permitido de itens que o atributo pode abrigar. O padrão é que cada atributo tenha um e somente um valor para o atributo. Quando um atributo for opcional ou quando puder ter mais do que uma ocorrência, a multiplicidade deve ser informada, indicando o valor mínimo e o valor máximo, da seguinte forma:

valor_mínimo .. valor_máximo

A seguir, são dados alguns exemplos:

- nome: String → instâncias da classe têm obrigatoriamente um e somente um nome.
- carteira: String [0..1] → instâncias da classe têm uma ou nenhuma carteira.
- telefones: Telefone [0..*] → instâncias da classe têm um ou vários telefones.
- pessoasContato: String [2] → instâncias da classe têm exatamente duas pessoas de contato.

Atributos podem ter um valor padrão inicial, ou seja, um valor que, quando não informado outro valor, será atribuído ao atributo. O campo *valorInicial* descreve exatamente este valor. O exemplo abaixo ilustra o uso de valor inicial.

origem: Ponto = (0,0) → a origem, quando não informado outro valor, será o ponto (0,0)

Finalmente, podem ser indicadas propriedades dos atributos. Uma propriedade que pode ser interessante mostrar em um modelo conceitual é a propriedade *readonly*, a qual indica que o valor do atributo não pode ser alterado após a inicialização do objeto. No exemplo abaixo, está-se indicando que o valor do atributo *numeroSocio* de um sócio de um clube não pode ser alterado.

numSocio: int {readonly}

Além das informações tratadas na declaração de um atributo seguindo a sintaxe da UML, outras informações de domínio, quando pertinentes, podem ser adicionadas no Dicionário de Dados do Projeto, tais como unidade de medida, intervalo de valores possíveis, limite, precisão etc.

Associações

Uma associação é um tipo de relacionamento que ocorre entre instâncias de duas ou mais classes. Assim como classes, associações são tipos. Ou seja, uma associação modela um tipo de relacionamento que pode ocorrer entre instâncias das classes envolvidas. Uma instância de uma associação (dita uma ligação) conecta instâncias específicas das classes envolvidas na associação. Seja o exemplo de um domínio em que clientes efetuam pedidos. Esse tipo de relacionamento pode ser modelado como uma associação *Cliente efetua Pedido*. Seja Pedro uma instância de *Cliente* e Pedido100 uma instância de *Pedido*. Se foi Pedro quem efetuou o Pedido100, então a ligação (Pedro, Pedido100) é uma instância da associação *Cliente efetua Pedido*.

Associações podem ser nomeadas. Neste texto sugere-se o uso de verbos conjugados, indicando o sentido de leitura. Ex.: Cliente (classe) *efetua* > (associação) Locação (classe). Cada classe envolvida na associação desempenha um papel, ao qual pode ser dado um nome. Cada classe envolvida na associação possui também uma

multiplicidade[14] nessa associação, que indica quantos objetos podem participar de uma instância dessa associação. A notação da UML tipicamente usada para representar associações em um modelo conceitual é ilustrada na Figura 6.3.
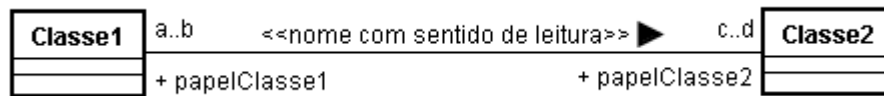
Figura 5.28 – Notação de Associações na UML.

Na ilustração da figura, um objeto da *Classe1* se relaciona com no mínimo *c* e no máximo *d* objetos da *Classe2*. Já um objeto da *Classe2* se relaciona com no mínimo *a* e no máximo *b* objetos da *Classe1*. Objetos da *Classe1* desempenham o papel de "*papelClasse1*" nesta associação, enquanto objetos da *Classe2* desempenham o papel de "*papelClasse2*" nessa mesma associação.

É importante, neste ponto, frisar a diferença entre sentido de leitura (ou direção do nome) de uma associação com a navegação da associação. O sentido de leitura diz apenas em que direção ler o nome da associação, mas nada diz sobre a navegabilidade da associação. A navegabilidade (linha de associação com seta direcionada) é usada para limitar a navegação de uma associação a uma única direção e é um recurso a ser usado apenas na fase de projeto. Em um modelo conceitual, todas as associações são não direcionais, ou seja, navegáveis nos dois sentidos.

Ainda que nomes de associações e papéis sejam opcionais, recomenda-se usá-los para tornar o modelo mais claro. Além disso, há algumas situações em que fica inviável ler um modelo se não houver a especificação do nome da associação ou de algum de seus papéis.

Seja o exemplo da Figura 5.29. Em uma empresa, um empregado está lotado em um departamento e, opcionalmente, pode chefiá-lo. Um departamento, por sua vez, pode ter vários empregados nele lotados, mas apenas um chefe. Sem nomear essas associações, o modelo fica confuso. Rotulando os papéis e as associações, o modelo torna-se muito mais claro. Na figura 5.29, um departamento exerce o papel de *departamento de lotação* do empregado e, neste caso, um empregado tem um e somente um departamento de lotação. No outro relacionamento, um empregado exerce o papel de *chefe* e, portanto, um departamento possui um e somente um chefe.
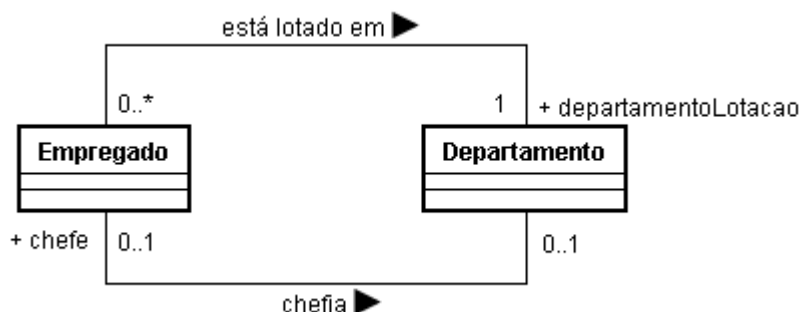
Figura 5.29 – Exemplo: Nomeando Associações.

[14] Multiplicidades em uma associação são análogas às multiplicidades em atributos e especificam as quantidades mínima e máxima de objetos que podem participar da associação. Quando nada for dito, o padrão é 1..1 como no caso de atributos. Contudo, para deixar os modelos claros, recomenda-se sempre especificar explicitamente as multiplicidades das associações.

Ao contrário das classes e dos atributos que podem ser encontrados facilmente a partir da leitura dos textos da descrição do minimundo e das descrições de casos de uso, muitas vezes, as informações sobre associações não aparecem tão explicitamente. Casos de uso descrevem ações de interação entre atores e sistema e, por isso, acabam mencionando principalmente operações. Operações transformam a informação, passando um objeto de um estado para outro, por meio da alteração dos seus valores de atributos e associações. Uma associação, por sua vez, é uma relação estática que pode existir entre duas classes. Assim, as descrições de casos de uso estão repletas de operações, mas não de associações (WAZLAWICK, 2004).

Contudo, conforme discutido na seção anterior, há alguns eventos que precisam ter sua ocorrência registrada e, portanto, são tipicamente mapeados como classes. Esses eventos estão descritos nos casos de uso e podem ter sido capturados como associações. Seja o exemplo de uma concessionária de automóveis. Neste domínio, clientes compram carros, como ilustra a parte (a) da Figura 5.30. Contudo, a compra é um evento importante para o negócio e precisa ser registrado. Neste caso, como ilustra a parte (b) da Figura 5.30, a compra deve ser tratada como uma classe e não como uma associação.
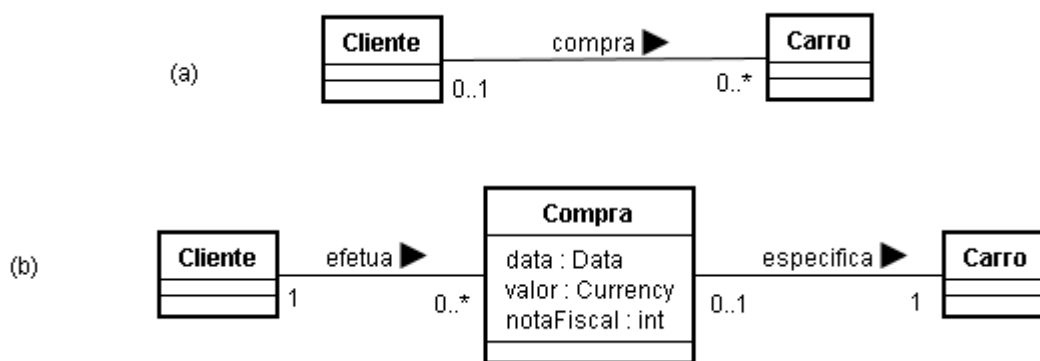
Figura 5.30 – Exemplo: Associação x Classe de Evento Lembrado.

Deve-se notar pelo exemplo acima que o evento é representado por uma classe, enquanto as associações continuam representando relacionamentos estáticos entre as classes e não operações ou transformações (WAZLAWICK, 2004). Assim, deve-se tomar cuidado com a representação de eventos como associações, questionando sempre se aquela associação é relevante para o sistema em questão.

Seja o exemplo da Figura 5.31. Nesse exemplo, o caso de uso aponta que funcionários são responsáveis por cadastrar livros em uma biblioteca. Seria necessário, pois, criar uma associação *Funcionário cadastra Livro* no modelo estrutural? A resposta, na maioria dos casos, é não. Apenas se explicitamente expresso pelo cliente em um requisito que é necessário saber exatamente qual funcionário fez o cadastro de um dado livro (o que é muito improvável de acontecer), é que tal relação deveria ser considerada. Mesmo se houver a necessidade de auditoria de uso do sistema (requisito não funcional relativo à segurança), não há a necessidade de modelar esta associação, pois requisitos não funcionais não devem ser considerados no modelo conceitual, uma vez que soluções bastante distintas à do uso dessa associação poderiam ser adotadas.
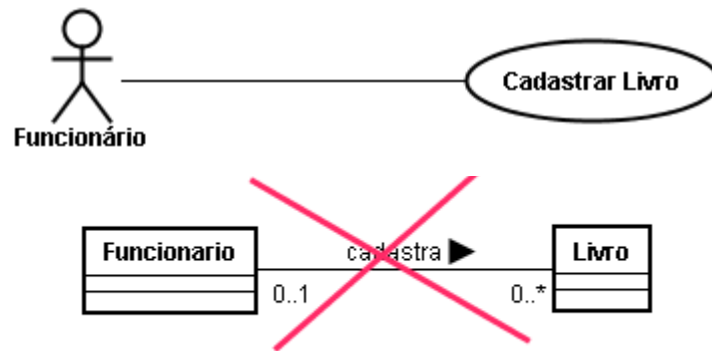
Figura 5.31 – Exemplo: Associação x Caso de Uso.

Na modelagem conceitual é fundamental saber a quantidade de objetos que uma associação admite em cada um de seus papéis, o que é capturado pelas multiplicidades da associação. Esta informação é bastante dependente da natureza do problema e do real significado da associação (o que se quer representar efetivamente), especialmente no que se refere à associação representar apenas o presente ou o histórico (WAZLAWICK, 2004).

Retomemos o exemplo da Figura 5.29, no qual se diz que um empregado está lotado em um departamento e, opcionalmente, pode chefiá-lo. Para definir precisamente as multiplicidades, é necessário investigar os seguintes aspectos: Um empregado pode mudar de lotação? Se sim, é necessário registrar apenas a lotação atual ou é necessário registrar o histórico de lotações dos empregados (ou seja, registrar o evento de lotação de um empregado em um departamento)? Um departamento pode, ao longo do tempo, mudar de chefe? Se sim, é necessário saber quem o histórico de chefias do departamento (ou seja, registrar o evento de nomeação do chefe do departamento)?

Como colocado no modelo da Figura 5.29, está-se representando apenas a situação presente. Se houver mudança de chefe de um departamento ou do departamento de lotação de um empregado, perder-se-á a informação histórica. Na maioria das vezes, essa não é uma solução aceitável. Na maioria dos domínios, as pessoas querem saber a informação histórica. Assim, nota-se que é parte das responsabilidades do sistema registrar a ocorrência dos eventos de nomeação do chefe e de lotação de empregados. Assim, um modelo mais fidedigno a essa realidade é o modelo da Figura 5.32, o qual introduz as classes do tipo “evento lembrado” *NomeacaoChefia* e *Lotacao*.
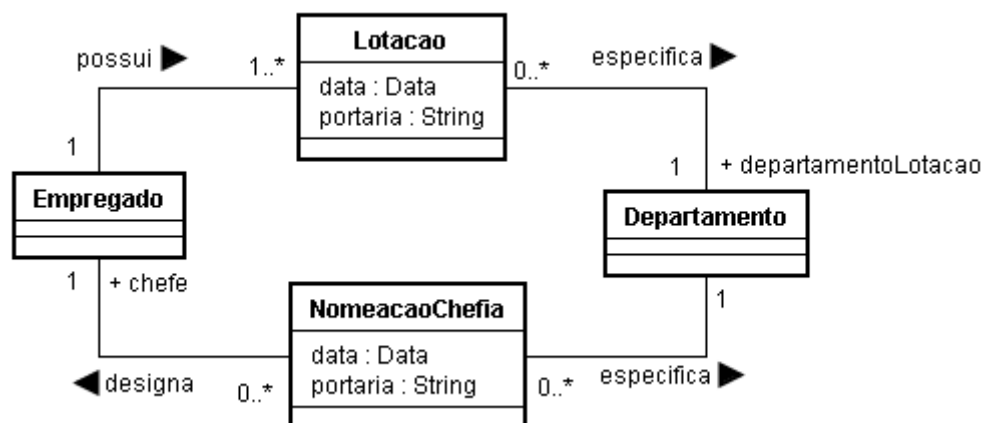
Figura 5.32– Registrando Históricos.

Ainda que este modelo seja mais fidedigno à realidade, ele ainda apresenta problemas. Por exemplo, o modelo diz que um empregado pode ter uma ou mais locações. Mas o empregado pode ter mais de uma lotação vigente? O mesmo vale para o caso da nomeação de chefia. Um empregado pode ser chefe de mais de um departamento ao mesmo tempo? Um departamento pode ter mais do que um chefe nomeado ao mesmo tempo? Infelizmente, o modelo é incapaz de responder a essas perguntas. Para eliminar essas ambiguidades, é necessário capturar regras de negócio do tipo restrições de integridade. No exemplo acima, as seguintes regras se aplicam:

- Um empregado só pode estar lotado em um único departamento em um dado momento.
- Um empregado só pode estar designado como chefe de um único departamento em um dado momento.
- Um departamento só pode ter um empregado designado como chefe em um dado momento.

Observe que, como um departamento pode ter vários empregados nele lotados ao mesmo tempo, não é necessário escrever uma restrição de integridade, pois este é o caso mais geral (sem restrição). Assim, restrições de integridade devem ser escritas apenas para as associações que são passíveis de restrições.

Restrições de integridade são regras de negócio e poderiam ser lançadas no Documento de Requisitos. Contudo, como elas são importantes para a compreensão e eliminação de ambiguidades do modelo conceitual, é útil descrevê-las no próprio modelo conceitual.

Além das restrições de integridade relativas às multiplicidades *n*, diversas outras restrições podem ser importantes para tornar o modelo mais fiel à realidade. Ainda no exemplo da Figura 5.32, poder-se-ia querer dizer que um empregado só pode ser nomeado como chefe de um departamento, se ele estiver lotado nesse departamento. Restrições deste tipo são comuns em porções fechadas de um diagrama de classes. Assim, toda vez que se detectar uma porção fechada em um diagrama de classes, vale a pena analisá-la para avaliar se há ali uma restrição de integridade ou não. Havendo, deve-se escrevê-la.

Restrições de integridade também podem falar sobre atributos das classes. Por exemplo, a data da portaria de nomeação de um empregado *e* como chefe de um departamento *d* deve ser igual ou posterior à data da portaria de lotação do empregado *e* no departamento *d*.

Geralmente, restrições de integridade são escritas em linguagem natural, uma vez que não são passíveis de modelagem gráfica. Contudo, conforme já discutido anteriormente, o uso de linguagem natural pode levar a ambiguidades. Visando suprir essa lacuna na UML, o OMG[15] incorporou ao padrão uma linguagem para especificação formal de restrições, a OCL (*Object Constraint Language*). Contudo, restrições escritas em OCL dificilmente serão entendidas por clientes e usuários, o que dificulta a validação das mesmas. Assim, neste texto, sugere-se escrever as restrições de integridade em linguagem natural mesmo.

[15] *Object Management Group* (http://www.omg.org/) é uma organização internacional que gerencia padrões abertos relativos ao desenvolvimento orientado a objetos, dentre eles a UML.

Vale ressaltar que a UML provê alguns mecanismos para representar restrições de integridade em um modelo gráfico. As próprias multiplicidades são uma forma de capturar restrições de integridade (ditas restrições de integridade de cardinalidade). Além das multiplicidades, a UML provê o recurso de restrições, as quais são representadas entre chaves (**{restrição}**). Restrições podem ser usadas, dentre outros, para restringir a ocorrência de associações. Seja o seguinte exemplo: em uma concessionária de automóveis compras podem ser financiadas ou por financeiras ou por bancos. Para capturar essa restrição, pode-se usar a restrição *xor* da UML, como ilustra a Figura 5.33.
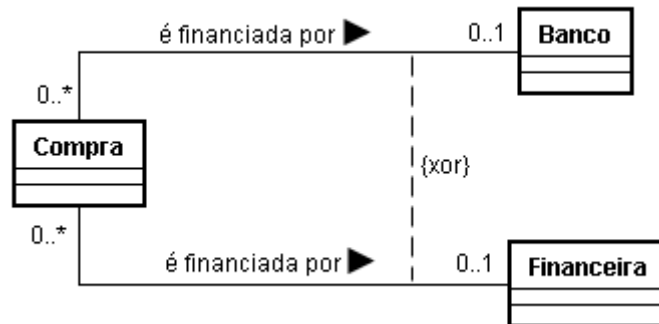
Figura 5.33 – Restrição XOR entre Associações.

Nesta figura, uma compra ou está relacionada a um banco ou a uma financeira. Não é possível que uma compra esteja associada aos dois ao mesmo. Como as multiplicidades mínimas do lado de banco e financeira são zero, uma compra pode não ser financiada.

Ainda em relação às multiplicidades, vale frisar que associações muitos-para-muitos são perfeitamente legais em um modelo orientado a objetos, como ilustra o exemplo da Figura 5.34. Nesse exemplo, está-se dizendo que disciplinas podem possuir vários pré-requisitos e podem ser pré-requisitos para várias outras disciplinas.
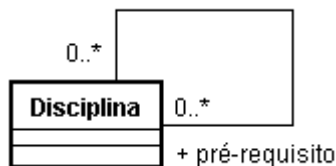
Figura 5.34 – Associação Muitos-para-Muitos.

Deve-se observar, no entanto, que muitas vezes, uma associação muitos-para-muitos oculta a necessidade de uma classe do tipo evento a ser lembrado. Seja o seguinte exemplo: em uma organização, empregados são alocados a projetos. Um empregado pode ser alocado a vários projeto, enquanto um projeto pode ter vários empregados a ele alocados. Tomando por base este fato, seria natural se chegar ao modelo da Figura 5.35(a). Contudo, se quisermos registrar as datas de início e fim do período em que o empregado esteve alocado ao projeto, esse modelo é insuficiente e deve ser alterado para comportar uma classe do tipo evento lembrado *Alocacao*, como mostra a Figura 5.35(b).
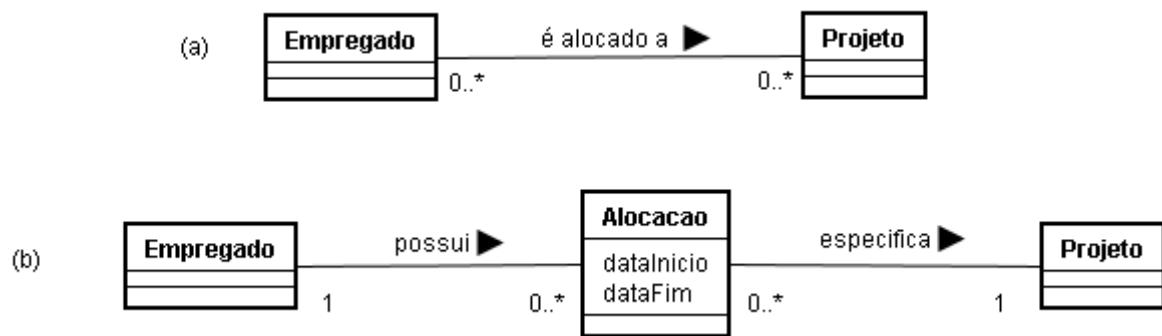
Figura 5.35– Associação Muitos-para-Muitos e Classes de Evento Lembrado.

De fato, o problema por detrás do modelo da Figura 5.35(a) é o mesmo anteriormente discutido na Figura 5.32: a necessidade ou não de se representar informação histórica. Contudo, de maneira mais abrangente, pode-se pensar que se uma associação apresenta atributos, é melhor tratá-la como uma nova classe. Seja o seguinte exemplo: em uma loja, um cliente efetua um pedido, discriminando vários produtos, cada um deles em uma certa quantidade. O modelo da Figura 5.36(a) procura representar essa situação, mas uma questão permanece em aberto: onde representar a informação da quantidade pedida de cada produto? Essa informação não pode ficar em *Produto*, pois diferentes pedidos pedem quantidades diferentes de um mesmo produto. Também não pode ficar em *Pedido*, pois um mesmo pedido tipicamente especifica diferentes quantidades de diferentes produtos. De fato, quantidade não é nem um atributo da classe *Pedido* nem um atributo da classe *Produto*, mas sim um atributo da associação *especifica*. Assim, uma solução possível é introduzir uma classe *ItemPedido*, reificando[16] essa associação, como ilustra a Figura 5.36(b).
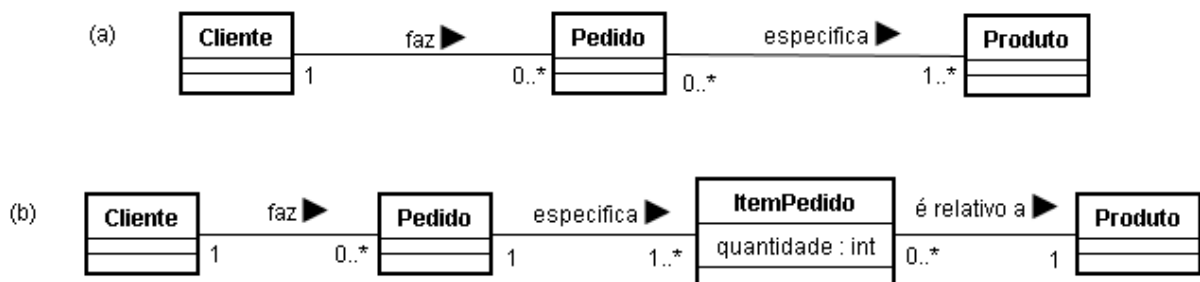
Figura 5.36 – Reificando uma Associação.

A UML oferece uma primitiva de modelagem, chamada *classe de associação*, que pode ser usada para tratar a reificação de associações (OLIVÉ, 2007). Uma classe de associação pode ser vista como uma associação que tem propriedades de classe (BOOCH; RUMBAUGH; JACOBSON, 2006). A Figura 6.12 mostra o exemplo anterior, sendo modelado como uma classe de associação, segundo a notação da UML.

[16] Reificar uma associação consiste em ver essa associação como uma classe. A palavra "reificação" vem da palavra do latim *res*, que significa coisa. Reificação corresponde ao que em linguagem natural se chama nominalização, que basicamente consiste em transformar um verbo em um substantivo (OLIVÉ, 2007).
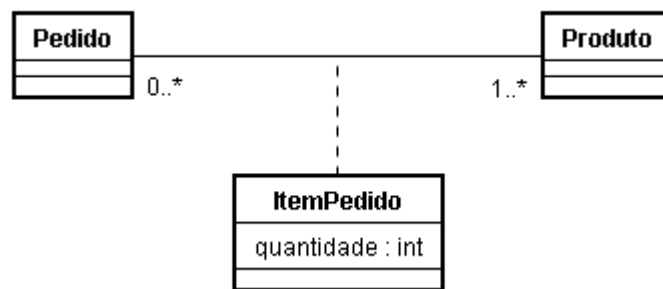
Figura 5.37 – Notação da UML para Classes Associativas.

Classes associativas são ainda representações de associações. Assim como uma instância de uma associação, uma instância de uma classe associativa é um par ordenado conectando duas instâncias das classes envolvidas na associação. Assim, se Pedido100 é uma instância de *Pedido*, Lápis é uma instância de *Produto* e o Pedido100 especifica 5 Lápis, então uma instância de *ItemPedido* é a tupla ((Pedido100, Lápis), 5).

Classes associativas podem ser usadas também para representar eventos cuja ocorrência precisa ser lembrada, como nos exemplos das figuras 5.32 e 5.35. Entretanto, é importante observar que o uso de classes associativas nesses casos pode levar a problemas de modelagem. Seja o seguinte contexto: em um hospital, pacientes são tratados em unidades médicas. Um paciente pode ser tratado em diversas unidades médicas diferentes, as quais podem abrigar diversos pacientes sendo tratados. A Figura 5.38(a) mostra um modelo que busca representar essa situação usando uma classe de associação. Como uma classe associativa, as instâncias de Tratamento são pares ordenados (Paciente, Unidade Médica). Assim, cada vez que um paciente é tratado em uma unidade médica diferente tem-se um tratamento. Essa pode, contudo, não ser precisamente a concepção do problema original. Poder-se-ia imaginar que um tratamento é um tratamento de um paciente em várias unidades médicas. A classe de associação não captura isso. Assim, um modelo mais fiel ao domínio é aquele que representa Tratamento como uma classe do tipo evento a ser lembrado e que está relacionada com Paciente e Unidade Médica da forma mostrada na Figura 5.38(b).
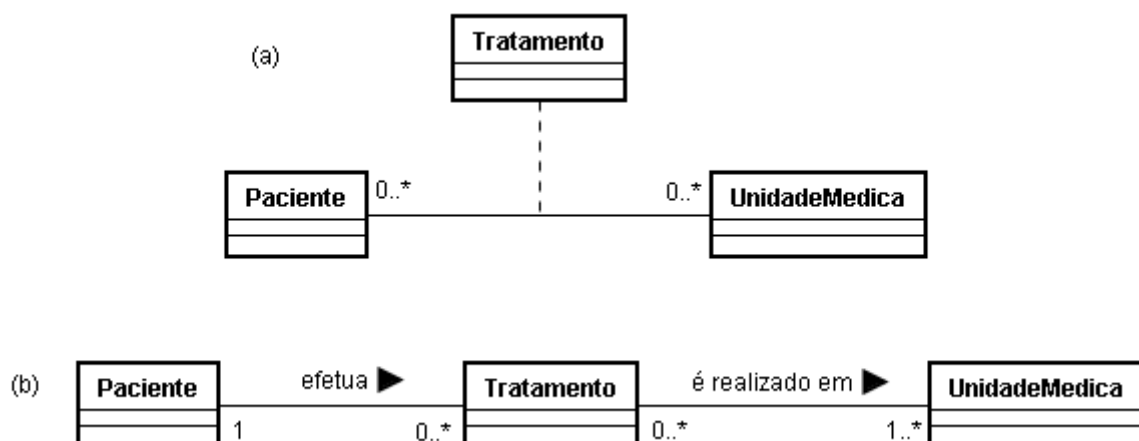
Figura 5.38 – Classes Associativas x Classes do Tipo Evento a Ser Lembrado.

Até o momento, todas as associações mostradas foram associações binárias, i.e., associações envolvendo duas classes. Mesmo o exemplo da Figura 5.34 (Disciplina tem como pré-requisito Disciplina) é ainda uma associação binária, na qual a mesma classe

desempenha dois papéis diferentes (disciplina que possui pré-requisito e disciplina que é pré-requisito). Entretanto, associações *n*-árias são também possíveis, ainda que bem menos corriqueiramente encontradas. Uma associação ternária, por exemplo, envolve três classes, como ilustra o exemplo da Figura 5.39. Nesse exemplo, está-se dizendo que fornecedores podem fornecer produtos para certos clientes.
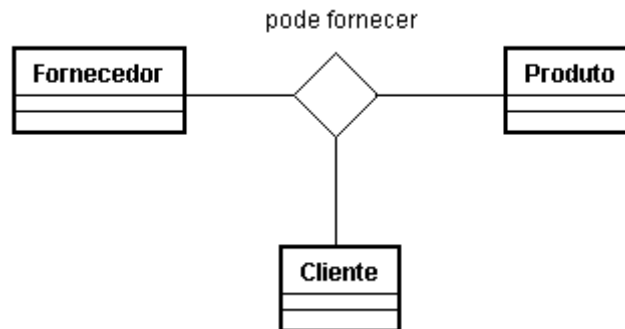
Figura 5.39 – Associação Ternária.

Na UML, associações *n*-árias são mostradas como losangos conectados às classes envolvidas na associação por meio de linhas sólidas, como mostra a Figura 5.39. O nome da associação é colocado dentro ou em cima do losango, sem direção de leitura. Normalmente, multiplicidades não são mostradas, dada a dificuldade de interpretá-las.

Finalmente, algumas associações podem ser consideradas mais fortes do que as outras, no sentido de que elas, na verdade, definem um objeto como sendo composto por outros (WAZLAWICK, 2004). Essas associações todo-parte podem ser de dois tipos principais: agregação e composição.

A *composição* é o tipo mais forte de associação todo-parte. Ela indica que um objeto-parte só pode ser parte de um único todo. Já a *agregação* não implica nessa exclusividade. Um carro, por exemplo, tem como partes um motor e quatro ou cinco rodas. Motor e rodas, ao serem partes de um carro, não podem ser partes de outros carros simultaneamente. Assim, esta é uma relação de composição, como ilustra a Figura 5.40(a). O exemplo da Figura 5.40(b) ilustra o caso de comissões compostas por professores. Nesse caso, um professor pode participar de mais de uma comissão simultaneamente e, portanto, trata-se uma relação de agregação. Na UML, um losango branco na extremidade da associação relativa ao todo indica uma agregação. Já um losango preto indica uma composição.
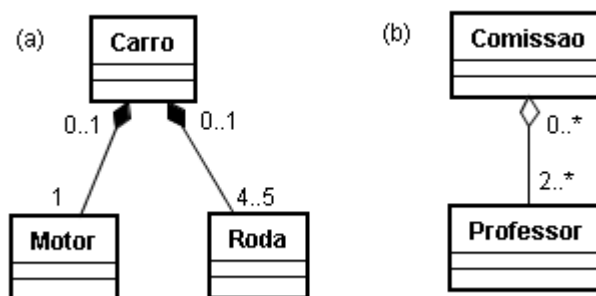
Figura 5.40 – Agregação e Composição.

Relações todo-parte podem ser empregadas em situações como:

- Quando há clareza de que um objeto complexo é composto de outros objetos (componente de). Ex.: Motor é um componente de um carro.
- Para designar membros de coleções (membro de). Ex.: Pesquisadores são membros de Grupos de Pesquisa.

Muitas vezes pode ser difícil perceber a diferença entre uma agregação / composição e uma associação comum. Quando houver essa dúvida, é melhor representar a situação usando uma associação comum, tendo em vista que ela impõe menos restrições.

5.5.3 – Especificação de Hierarquias de Generalização / Especialização

Um dos principais mecanismos de estruturação de conceitos é a generalização / especialização. Com este mecanismo é possível capturar similaridades entre classes, dispondo-as em hierarquias de classes. No contexto da orientação a objetos, esse tipo de relacionamento é também conhecido como herança.

É importante notar que a herança tem uma natureza bastante diferente das associações. Associações representam possíveis ligações entre instâncias das classes envolvidas. Já a relação de herança é uma relação entre classes e não entre instâncias. Ao se considerar uma classe *B* como sendo uma subclasse de outra classe *A* está-se assumindo que todas as instâncias de *B* são também instâncias de *A*. Assim, ao se dizer que a classe *EstudanteGraduacao* herda da classe *Estudante*, está-se indicando que todos os estudantes de graduação são estudantes. Em resumo, deve-se interpretar a relação de herança como uma relação de subtipo entre classes. A Figura 5.41 mostra a notação da UML para representar herança.
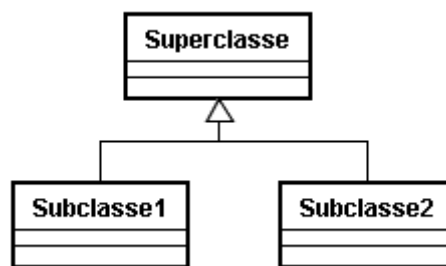
Figura 5.41 – Notação de Herança da UML.

A relação de herança é aplicável quando for necessário fatorar os elementos de informações (atributos e associações) de uma classe. Quando um conjunto de classes possuir semelhanças e diferenças, então elas podem ser organizadas em uma hierarquia de classes, de forma a agrupar em uma superclasse os elementos de informação comuns, deixando as especificidades nas subclasses.

De maneira geral, não faz sentido criar hierarquias de classes, quando as especializações (subclasses) não tiverem nenhum elemento de informação diferente. Quando isso ocorrer, é normalmente suficiente criar um atributo *tipo*, para indicar os possíveis subtipos da generalização. Seja o caso de um domínio em que se faz distinção entre clientes normais e clientes especiais, dos quais se quer saber exatamente as mesmas informações. Neste caso, criar uma hierarquia de classes, como ilustra a Figura

5.42(a), é desnecessário. Uma solução como a apresentada na Figura 5.42(b), em que o atributo *tipo* pode ser de um tipo enumerado com os seguintes valores {Normal, Especial}, modela satisfatoriamente o problema e é mais simples e, portanto, mais indicada.
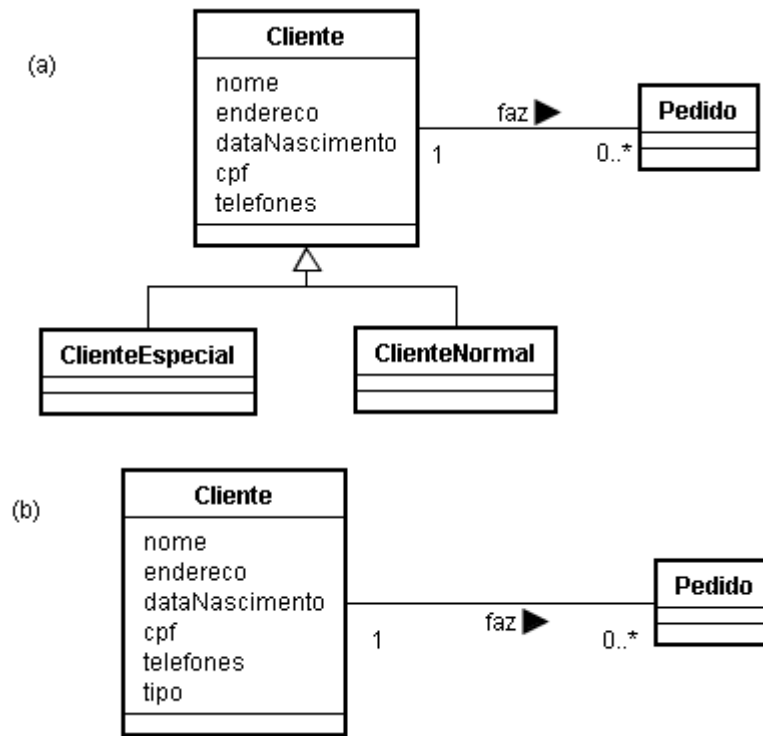
Figura 5.42 – Uso ou não de Herança.

Também não faz sentido criar uma hierarquia de classes em que a superclasse não tem nenhum atributo ou associação. Informações de estados pelos quais um objeto passa também não devem ser confundidas com subclasses. Por exemplo, um carro de uma locadora de automóveis pode estar locado, disponível ou em manutenção. Estes são estados e não subtipos de carro.

De fato, é interessante considerar alguns critérios para incluir uma subclasse (ou superclasse) em um modelo conceitual. O principal deles é o fato da especialização (ou generalização) estar dentro do domínio de responsabilidade do sistema. Apenas subclasses (superclasses) relevantes para o sistema em questão devem ser consideradas. Além desse critério básico, os seguintes critérios devem ser usados para analisar hierarquias de herança:

- Uma hierarquia de classes deve modelar relações “é-um-tipo-de”, ou seja, toda subclasse deve ser um subtipo específico de sua superclasse.
- Uma subclasse deve possuir todas as propriedades (atributos e associações) definidas por suas superclasses e adicionar mais alguma coisa (algum outro atributo ou associação).
- Todas as instâncias de uma subclasse têm de ser também instâncias da superclasse.

Atenção especial deve ser dada à nomeação de classes em uma hierarquia de classes. Cada especialização deve ser nomeada de forma a ser auto-explicativa. Um nome apropriado para a especialização pode ser formado pelo nome de sua superclasse,

acompanhado por um qualificador que descreve a natureza da especialização. Por exemplo, *EstudanteGraduacao* para designar um subtipo de *Estudante*.

Hierarquias de classes não devem ser usadas de forma não criteriosa, simplesmente para compartilhar algumas propriedades. Seja o caso de uma loja de animais, em que se deseja saber as seguintes informações sobre clientes e animais: nome, data de nascimento e endereço. Não faz nenhum sentido considerar que *Cliente* é uma subclasse de *Animal* ou vice-versa, apenas para reusar um conjunto de atributos que, coincidentemente, é igual.

No que se refere à modelagem de superclasses, deve-se observar se uma superclasse é *concreta* ou *abstrata*. Se a superclasse puder ter instâncias próprias, que não são instâncias de nenhuma de suas subclasses, então ela é uma classe concreta. Por outro lado, se não for possível instanciar diretamente a superclasse, ou seja, se todas as instâncias da superclasse são antes instâncias das suas subclasses, então a superclasse é abstrata. Classes abstratas são representadas na UML com seu nome escrito em itálico e não devem herdar de classes concretas.

Quando modeladas hierarquias de herança, é necessário posicionar atributos e associações adequadamente. Cada atributo ou associação deve ser colocado na classe mais adequada. Atributos e associações genéricos, que se aplicam a todas as subclasses, devem ser posicionados no topo da estrutura, de modo a serem aplicáveis a todas as especializações. De maneira mais geral, se um atributo ou associação é aplicável a um nível inteiro de especializações, então ele deve ser posicionado na generalização correspondente. Por outro lado, se algumas vezes um atributo ou associação tiver um valor significativo, mas em outras ele não for aplicável, deve-se rever seu posicionamento ou mesmo a estrutura de generalização-especialização adotada.

Inevitavelmente, o processo detalhado de designar atributos e associações a classes conduz a um entendimento mais completo da hierarquia de herança do que era possível em um estágio anterior. Assim, deve-se esperar que o trabalho de reposicionamento de atributos e associações conduza a uma revisão de uma hierarquia de classes.

Por fim vale a pena mencionar que, durante anos, o mecanismo de herança foi considerado o grande diferencial da orientação a objetos. Contudo, com o passar do tempo, essa ênfase foi perdendo força, pois se percebeu que o uso da herança nem sempre conduz à melhor solução de um problema de modelagem. Hoje a herança é considerada apenas mais uma ferramenta de modelagem, utilizada basicamente para fatorar informações, as quais, de outra forma, ficariam repetidas em diferentes classes (WAZLAWICK, 2004).

5.6 - Modelagem Dinâmica

Um sistema de informação realiza ações. O efeito de uma ação pode ser uma alteração em sua base de informação e/ou a comunicação de alguma informação ou comando para um ou mais destinatários. Um evento de requisição de ação (ou simplesmente uma requisição) é uma solicitação para o sistema realizar uma ação. O esquema comportamental de um sistema visa especificar essas ações (OLIVÉ, 2007).

Uma parte importante do modelo comportamental de um sistema é o modelo de casos de uso, o qual fornece uma visão das funcionalidades que o sistema deve prover. O modelo conceitual estrutural define os tipos de entidades (classes) e de

relacionamentos (atributos e associações) do domínio do problema que o sistema deve representar para poder prover as funcionalidades descritas no modelo de casos de uso. Durante a realização de um caso de uso, atores geram eventos de requisição de ações para o sistema, solicitando a execução de alguma ação. O sistema realiza ações e ele próprio pode gerar outras requisições de ação. É necessário, pois, modelar essas requisições de ações, as correspondentes ações a serem realizadas pelo sistema e seus efeitos. Este é o propósito da modelagem dinâmica.

Em uma abordagem orientada a objetos, requisições de ação correspondem a mensagens trocadas entre objetos. As ações propriamente ditas e seus efeitos são tratados pelas operações das classes[17]. Assim, a modelagem dinâmica está relacionada com as trocas de mensagens entre objetos e a modelagem das operações das classes.

Os diagramas de classes gerados pela atividade de modelagem conceitual estrutural representam apenas os elementos estáticos de um modelo de análise orientada a objetos. É preciso, ainda, modelar o comportamento dinâmico da aplicação. Para tal, é necessário representar o comportamento do sistema como uma função do tempo e de eventos específicos. Um modelo de dinâmico indica como o sistema irá responder a eventos ou estímulos externos e auxilia o processo de descoberta das operações das classes do sistema.

Para apoiar a modelagem da dinâmica de sistemas, a UML oferece três tipos de diagramas (BOOCH; RUMBAUGH; JACOBSON, 2006): *Diagrama de Gráfico de Estados, Diagrama de Interação* e *Diagrama de Atividades.* Neste material é discutida apenas a modelagem de estados.

5.6.1 – Diagrama de Estados

Um diagrama de estados mostra uma máquina de estados que consiste dos estados pelos quais objetos de uma particular classe podem passar ao longo de seu ciclo de vida e as transições possíveis entre esses estados, as quais são resultados de eventos que atingem esses objetos. Diagramas de gráfico de estados (ou **diagramas de transição de estados**) são usados principalmente para modelar o comportamento de uma classe, dando ênfase ao comportamento específico de seus objetos.

Classes com estados (ou modais) são classes cujas instâncias podem mudar de um estado para outro ao longo de sua existência, mudando possivelmente sua estrutura, seus valores de atributos ou comportamento dos métodos (WAZLAWICK, 2004).

Classes modais podem ser modeladas como máquinas de estados finitos. Uma máquina de estados finitos é uma máquina que, em um dado momento, está em um e somente um de um número finito de estados (OLIVÉ, 2007). Os estados de uma máquina de estados de uma classe modal correspondem às situações relevantes em que as instâncias dessa classe podem estar durante sua existência. Um estado é considerado relevante quando ele ajuda a definir restrições ou efeitos dos eventos.

Em qualquer estado, uma máquina de estados pode receber estímulos. Quando a máquina recebe um estímulo, ela pode realizar uma transição de seu estado corrente (dito estado origem) para um outro estado (dito estado destino), sendo que se assume

[17] De fato, abordagens distintas podem ser usadas, tal como representar tipos de requisições como classes, ditas classes de evento, e os seus efeitos como operações das correspondentes classes de evento, tal como faz Olivé (2007).

que as transições são instantâneas. A definição do estado destino depende do estado origem e do estímulo recebido. Além disso, os estado origem e destino em uma transição podem ser o mesmo. Neste caso, a transição é dita uma autotransição (OLIVÉ, 2007).

Diagramas de Transições de Estados são usados modelar o comportamento de instâncias de uma classe modal como uma máquina de estados. Todas as instâncias da classe comportam-se da mesma maneira. Em outras palavras, cada diagrama de estados é construído para uma única classe, com o objetivo de mostrar o comportamento ao longo do tempo de vida de seus objetos. Diagramas de estados descrevem os possíveis estados pelos quais objetos da classe podem passar e as alterações dos estados como resultado de eventos (estímulos) que atingem esses objetos. Uma máquina de estado especifica a ordem válida dos estados pelos quais os objetos da classe podem passar ao longo de seu ciclo de vida. A Figura 5.43 mostra a notação básica da UML para diagramas de gráfico de estados.
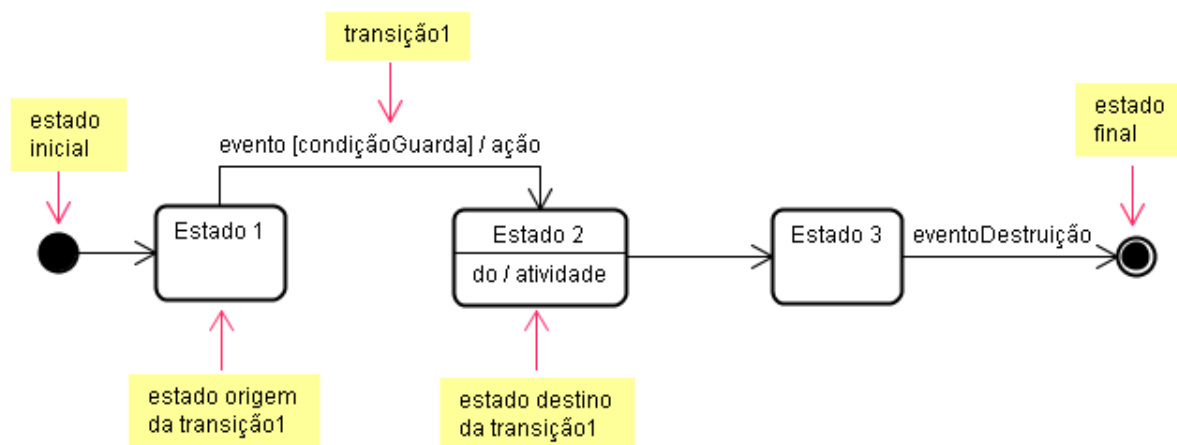
Figura 5.43 - Notação Básica da UML para Diagramas de Gráfico de Estados.

Um estado é uma situação na vida de um objeto durante a qual o objeto satisfaz alguma condição, realiza alguma atividade ou aguarda a ocorrência de um evento (BOOCH; RUMBAUGH; JACOBSON, 2006). Estados são representados por retângulos com os cantos arredondados, sendo que o nome de um estado deve ser único em uma máquina de estados. Uma regra prática para nomear estados consiste em atribuir um nome tal que sejam significativas sentenças do tipo "*o <<objeto>> está <<nome do estado>>*" ou "*o <<objeto>> está no estado <<nome do estado>>*". Por exemplo, em um sistema de locadora de automóveis, um estado possível de objetos da classe *Carro* seria "Disponível". A sentença "o carro está disponível" tem um significado claro (OLIVÉ, 2007).

Quando um objeto fica realizando uma atividade durante todo o tempo em que permanece em um certo estado, deve-se indicar essa atividade no compartimento de ações do respectivo estado. É importante realçar que uma atividade tem duração significativa e, quando concluída, tipicamente a conclusão provoca uma transição para um novo estado. A notação da UML para representar atividades de um estado é: **do / <<nomeAtividade>>**.

Transições são representadas por meio de setas rotuladas. Uma transição envolve um estado origem, um estado destino e normalmente um evento, dito o gatilho da transição. Quando a máquina de estados se encontra no estado origem e recebe o evento

gatilho, então o evento dispara a transição e a máquina de estados vai para o estado destino. Se uma máquina recebe um evento que não é um gatilho para nenhuma transição, então ela não é afetada pelo evento (OLIVÉ, 2007).

Uma transição pode ter uma condição de guarda associada. Às vezes, há duas ou mais transições com o mesmo estado origem e o mesmo evento gatilho, mas com condições de guarda diferentes. Neste caso, a transição é disparada somente quando o evento gatilho ocorre e a condição de guarda é verdadeira (OLIVÉ, 2007). Quando uma transição não possuir uma condição de guarda associada, então ela ocorrerá sempre que o evento ocorrer.

Por fim, quando uma transição é disparada, uma ação instantânea pode ser realizada. Assim, o rótulo de uma transição pode ter até três partes, todas elas opcionais:

evento [condiçãoGuarda] / ação

Basicamente, a semântica de um diagrama de estados é a seguinte: quando o *evento* ocorre, se a *condição de guarda* é verdadeira, a *transição* dispara e a *ação* é realizada instantaneamente. O objeto passa, então, do *estado origem* para o *estado destino*. Se o estado destino possuir uma *atividade* a ser realizada, ela é iniciada.

O fato de uma transição não possuir um evento associado normalmente aponta para a existência de um evento implícito. Isso tipicamente ocorre em três situações: (i) o evento implícito é a conclusão da atividade do estado origem e a transição ocorrerá tão logo a atividade associada ao estado origem tiver sido concluída; (ii) o evento implícito é temporal, sendo disparado pela passagem do tempo; (iii) o evento implícito torna a condição de guarda verdadeira na base de informações do sistema, mas o evento em si não é modelado.

Embora ambos os termos ação e atividade denotem processos, eles não devem ser confundidos. Ações são consideradas processos instantâneos; atividades, por sua vez, estão sempre associadas a estados e têm duração no tempo. Vale a pena observar que, no mundo real, não há processos efetivamente instantâneos. Por mais rápida que seja, uma ação ocorrerá sempre em um intervalo de tempo. Esta simplificação de se considerar ações instantâneas no modelo conceitual pode ser associada à ideia de que a ação ocorre tão rapidamente que não é possível interrompê-la. Em contraste, uma atividade é passível de interrupção, sendo possível, por exemplo, que um evento ocorra, interrompa a atividade e provoque uma mudança no estado do objeto antes da conclusão da atividade.

Às vezes quer se modelar situações em que uma ação instantânea é realizada quando se entra ou sai de um estado, qualquer que seja a transição que o leve ou o retire desse estado. Seja o exemplo de um elevador. Neste contexto, ao parar em um certo andar, o elevador abre a porta. Suponha que a abertura da porta do elevador seja um processo que não possa ser interrompido e, portanto, que se opte por modelá-lo como uma ação. Essa ação deverá ocorrer sempre que o elevador entrar no estado "*Parado*" e deve ser indicada no compartimento de ações desse estado como sendo uma ação de entrada no estado. A notação da UML para representar ações de entrada em um estado é: **entry / <<nomeAção>>**. Para representar ações de saída de um estado a notação é: **exit / <<nomeAção>>**.

Restam ainda na Figura 5.43 dois tipos especiais de estados: os ditos estados inicial e final. Conforme citado anteriormente, um objeto está sempre em um e somente um estado. Isso implica que, ao ser instanciado, o objeto precisa estar em algum estado.

O estado inicial é precisamente esse estado. Graficamente, um estado inicial é mostrado como um pequeno círculo preenchido na cor preta. Seu significado é o seguinte: quando o objeto é criado, ele é colocado no estado inicial e sua transição de saída é automaticamente disparada, movendo o objeto para um dos estados da máquina de estados (no caso da Figura 5.43, para o *Estado1*). Toda máquina de estados tem de ter um (e somente um) estado inicial. Note que o estado inicial não se comporta como um estado normal[18], uma vez que objetos não se mantêm nele por um período de tempo. Ao contrário, uma vez que eles entram no estado inicial, sua transição de saída é imediatamente disparada e o estado inicial é abandonado. A transição de saída do estado inicial tem como evento gatilho implícito o evento responsável pela criação do objeto (OLIVÉ, 2007) e, na UML, esse evento não é explicitamente representado. Estados iniciais têm apenas transições de saída. As transições de saída de um estado inicial podem ter condições de guarda e/ou ações associadas. Quando houver condições de guarda, deve-se garantir que sempre pelo menos uma das transições de saída poderá ser disparada.

Quando um objeto deixa de existir, obviamente ele deixa de estar em qualquer um dos estados. Isso pode ser dito no diagrama por meio de uma transição para o estado final. O estado final indica, na verdade, que o objeto deixou de existir. Na UML um estado final é representado como um círculo preto preenchido com outro círculo não preenchido ao seu redor, como mostra a Figura 5.43. As transições para o estado final definem os estados em que é possível excluir o objeto. Classes cujos objetos não podem ser excluídos, portanto, não possuem um estado final (OLIVÉ, 2007). Assim como o estado inicial, o estado final não se comporta como um estado normal, uma vez que o objeto também não permanece nesse estado (já que o objeto não existe mais). Ao contrário do estado inicial, contudo, uma máquina de estados pode ter vários estados finais. Além disso, deve-se representar o evento que elimina o objeto (na Figura 5.43, *eventoDestruição*).

É importante indicar no diagrama de estados os eventos maiores (eventos de domínio e requisições de ações) e não os eventos estruturais que efetivamente alteram o estado do objeto. Assim, neste texto sugere-se indicar como eventos de transições de uma máquina de estados as requisições de realização de casos de uso do sistema (ou de fluxos de eventos específicos, quando um caso de uso tiver mais de um fluxo de eventos normal). Para facilitar a rastreabilidade, sugere-se usar como nome do evento exatamente o mesmo nome do caso de uso (ou do fluxo de eventos). Seja o exemplo de uma locadora de automóveis, que possua, dentre outros, os casos de uso mostrados na Figura 5.44, os quais possuem os fluxos de eventos mostrados nas notas anexadas aos casos de uso.

[18] Por não se comportar como um estado normal, o estado inicial é considerado um pseudoestado no metamodelo da UML.
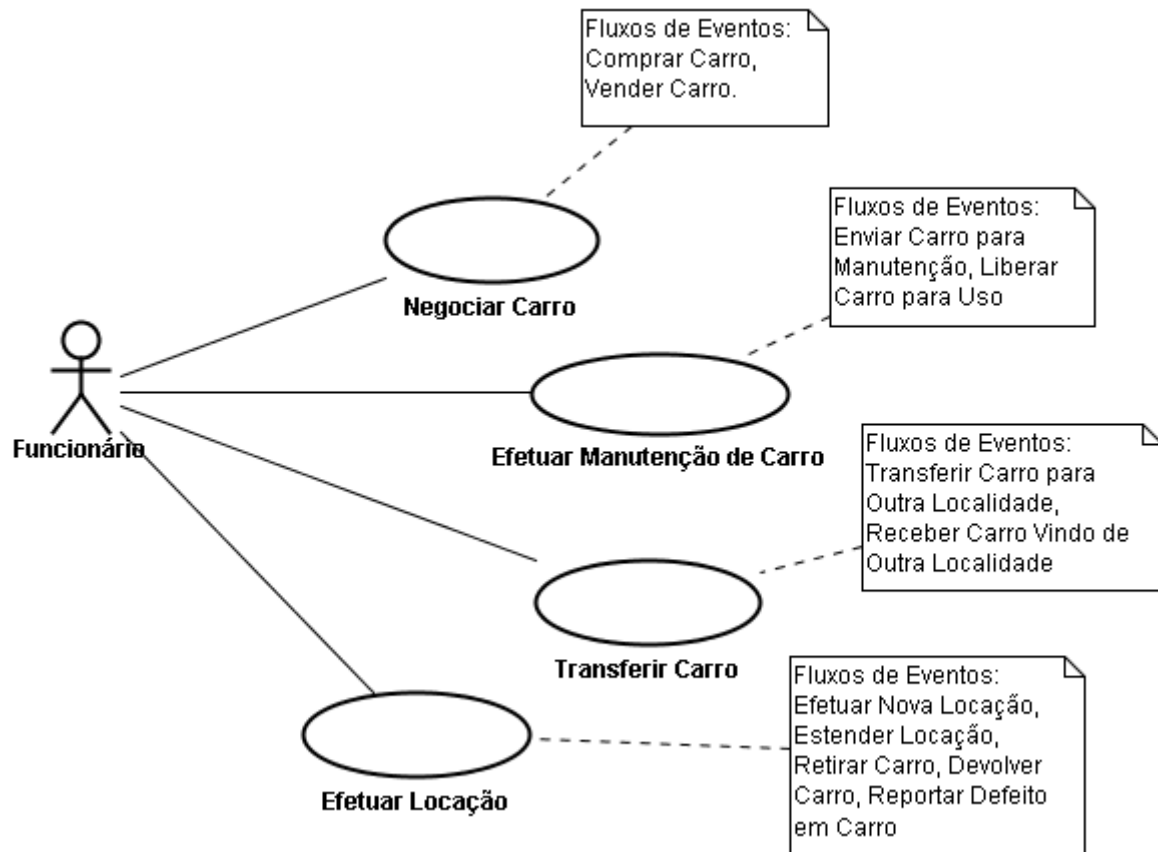
Figura 5.44 – Locadora de Automóveis - Casos de Uso e Fluxos de Eventos Associados.

A classe *Carro* tem o seu comportamento definido pela máquina de estados do diagrama de gráfico de estados da Figura 5.45. Ao ser adquirido (fluxo de eventos *Comprar Carro*, do caso de uso *Negociar Carro*), o carro é colocado *Em Preparação*. Quando liberado para uso (fluxo de eventos *Liberar Carro para Uso*, do caso de uso *Efetuar Manutenção de Carro*), o carro fica *Disponível*. Quando o cliente retira o carro (fluxo de eventos *Retirar Carro*, do caso de uso *Efetuar Locação*), este fica *Em Uso*. Quando é devolvido (fluxo de eventos *Devolver Carro*, do caso de uso *Efetuar Locação*), o carro fica novamente *Em Preparação*. Quando *Disponível*, um carro pode ser transferido de uma localidade para outra (fluxo de eventos *Transferir Carro para Outra Localidade* do caso de uso *Transferir Carro*). Durante o trânsito de uma localidade para outra, o carro está *Em Trânsito*, até ser recebido na localidade destino (fluxo de eventos *Receber Carro Vindo de Outra Localidade*, do caso de uso *Transferir Carro*), quando novamente é colocado *Em Preparação*. Finalmente, carros *Em Preparação* podem ser vendidos (fluxo de eventos *Vender Carro*, do caso de uso *Negociar Carro*), quando deixam de pertencer à locadora e são eliminados de sua base de informações.
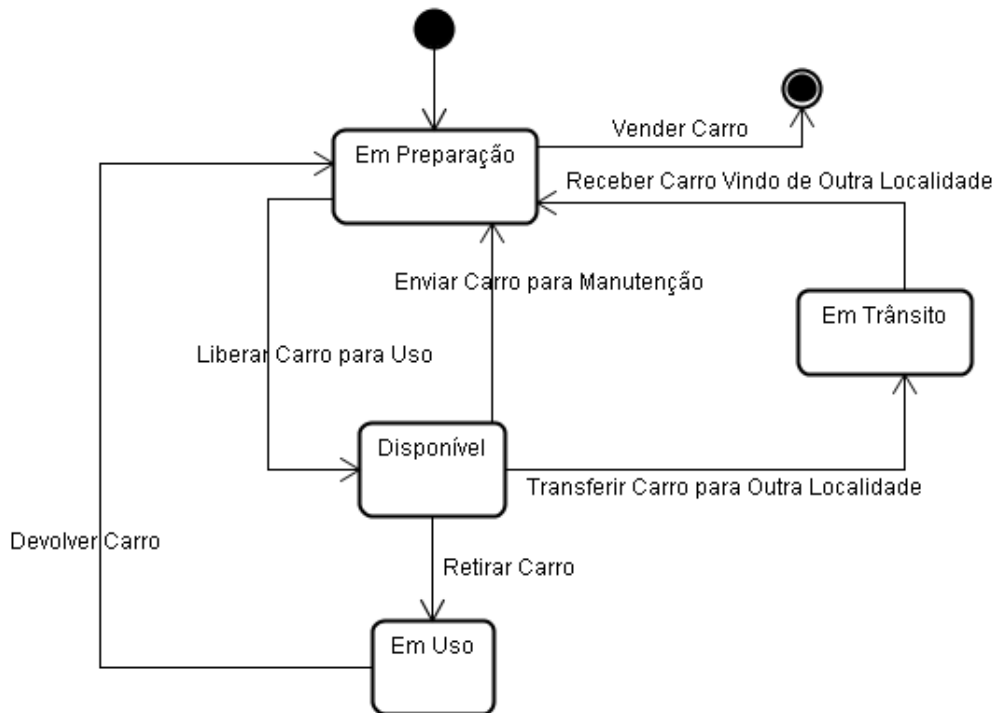
Figura 5.45 – Diagrama de Gráfico de Estados da Classe *Carro* – Disponibilidade (adaptado de (OLIVÉ, 2007)).

Nem todas as classes precisam ser modeladas como máquinas de estados. Apenas classes modais, i.e., que apresentam comportamento variável em função do estado de seus objetos, necessitam ser modeladas como máquinas de estados. Além disso, para os diagramas de estados serem efetivamente úteis, recomenda-se modelar uma máquina de estados somente se a classe em questão tiver três ou mais estados relevantes. Se uma classe possuir apenas dois estados relevantes, ainda cabe desenvolver uma máquina de estados. Contudo, de maneira geral, o diagrama tende a ser muito simples e a acrescentar pouca informação relevante que justifique o esforço de elaboração e manutenção do correspondente diagrama. Neste caso, os estados e transições podem ser levantados, sem no entanto elaborar um diagrama de estados.

Para algumas classes, pode ser útil desenvolver mais do que um diagrama de estados, cada qual modelando o comportamento dos objetos da classe por uma perspectiva diferente. Em um determinado momento, um objeto está em um (e somente um) estado em cada uma de suas máquinas de estado. Cada diagrama define seu próprio conjunto de estados nos quais um objeto pode estar, a partir de diferentes pontos de vista (OLIVÉ, 2007). Seja novamente o exemplo da classe *Carro*. A Figura 5.45 mostra os possíveis estados de um carro segundo um ponto de vista de disponibilidade. Entretanto, independentemente da disponibilidade, do ponto de vista de negociabilidade, um carro pode estar em dois estados (Não à Venda, À Venda), como mostra a Figura 5.46.

Vale ressaltar que os diferentes diagramas de estados de uma mesma classe não devem ter estados comuns. Cada diagrama deve ter seu próprio conjunto de estados e cada estado pertence a somente um diagrama de estados. Já os eventos podem aparecer em diferentes diagramas de estados, inclusive de classes diferentes. Quando um evento aparecer em mais de um diagrama de estados, sua ocorrência vai disparar as

correspondentes transições em cada uma das máquinas de estados em que ele aparecer (OLIVÉ, 2007).
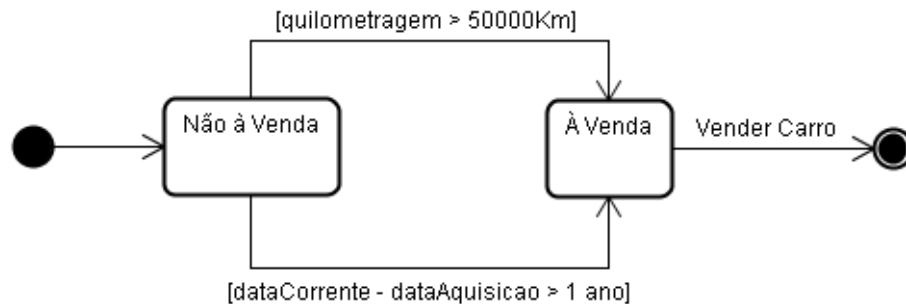
Figura 5.46 – Diagrama de Gráfico de Estados da Classe *Carro* – Negociabilidade (adaptado de (OLIVÉ, 2007)).

A Figura 5.46 mostra duas transições em que os eventos não são declarados explicitamente. No primeiro caso (quilometragem > 50000Km), o evento implícito torna a condição de guarda verdadeira na base de informações do sistema. Esse evento corresponde ao registro no sistema de qual é a quilometragem corrente do carro. Caso esse registro ocorra sempre no ato da devolução do carro pelo cliente (fluxo de eventos *Devolver Carro*, do caso de uso *Efetuar Locação*) e/ou no ato do recebimento do carro vindo de outra localidade (fluxo de eventos *Receber Carro Vindo de Outra Localidade*, do caso de uso *Transferir Carro*), esses eventos poderiam ser explicitamente declarados. Contudo, se o registro pode ocorrer em vários eventos diferentes, é melhor deixar o evento implícito. O segundo caso (dataCorrente – dataAquisicao > 1 ano) trata-se de um evento temporal, disparado pela passagem do tempo.

Todos os estados mostrados até então são estados simples, i.e., estados que não possuem subestados. Entretanto, há também estados compostos, os quais podem ser decompostos em um conjunto de subestados disjuntos e mutuamente exclusivos e um conjunto de transições (OLIVÉ, 2007). Um subestado é um estado aninhado em outro estado. O uso de estados compostos e subestados é bastante útil para simplificar a modelagem de comportamentos complexos. Seja o exemplo da Figura 5.45, que trata da disponibilidade de um carro. Suponha que seja necessário distinguir três subestados do estado *Em Uso*, a saber: *Em Uso Normal*, quando o carro não está quebrado nem em atraso; *Quebrado*, quando o cliente reportar um defeito no carro; e *Em Atraso*, quando o carro não foi devolvido na data de devolução prevista e não está quebrado. A Figura 5.47 mostra a máquina de estados da classe *Carro* considerando, agora, que, quando um carro está em uso, ele pode estar nesses três subestados.
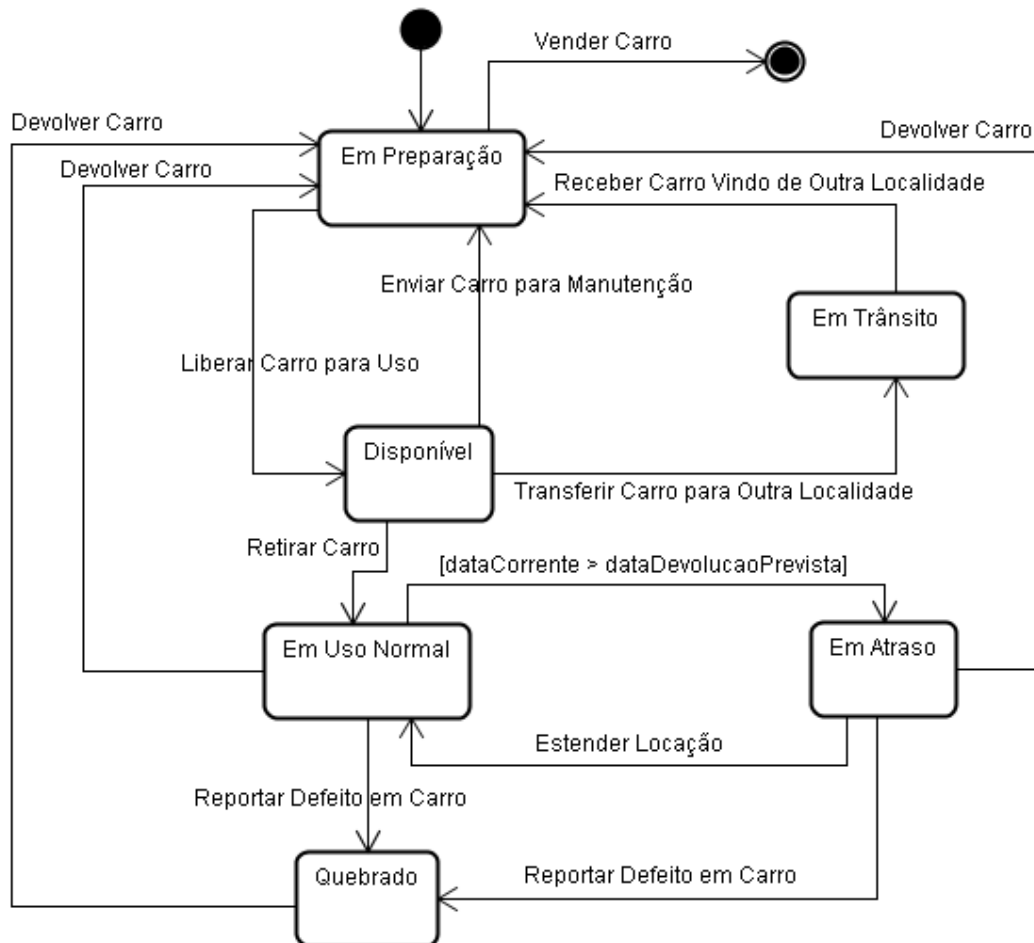
Figura 5.47 – Diagrama de Estados da Classe *Carro* (Disponibilidade) com Subestados de *Em Uso* (adaptado de (OLIVÉ, 2007)).

Nesse diagrama, não está sendo mostrado que os estados *Em Uso Normal*, *Em Atraso* e *Quebrado* são, de fato, subestados do estado *Em Uso* e, portanto, transições comuns (por exemplo, aquelas provocadas pelo evento *Devolver Carro*) são repetidas. Isso torna o modelo mais complexo e fica claro que esta solução representando diretamente os subestados (e omitindo o estado composto) não é escalável para sistemas que possuem muitos subestados, levando a diagramas confusos e desestruturados (OLIVÉ, 2007). A Figura 5.48 mostra uma solução mais indicada, em que tanto o estado composto quanto seus subestados são mostrados no mesmo diagrama. Uma outra opção é ocultar a decomposição do estado composto, mantendo o diagrama como o mostrado na Figura 5.45, e mostrar essa decomposição em um diagrama de estados separado.

Se um objeto está em um estado composto, então ele deve estar também em um de seus subestados. Assim, um estado composto pode possuir um estado inicial para indicar o subestado padrão do estado composto, como representado na Figura 5.48. Entretanto, deve-se considerar que as transições podem começar e terminar em qualquer nível. Ou seja, uma transição pode ir (ou partir) diretamente de um subestado (OLIVÉ, 2007). Assim, uma outra opção para o diagrama da Figura 5.48 seria fazer a transição nomeada pelo evento *Retirar Carro* chegar diretamente ao subestado *Em Uso Normal*, ao invés de chegar ao estado composto *Em Uso*.
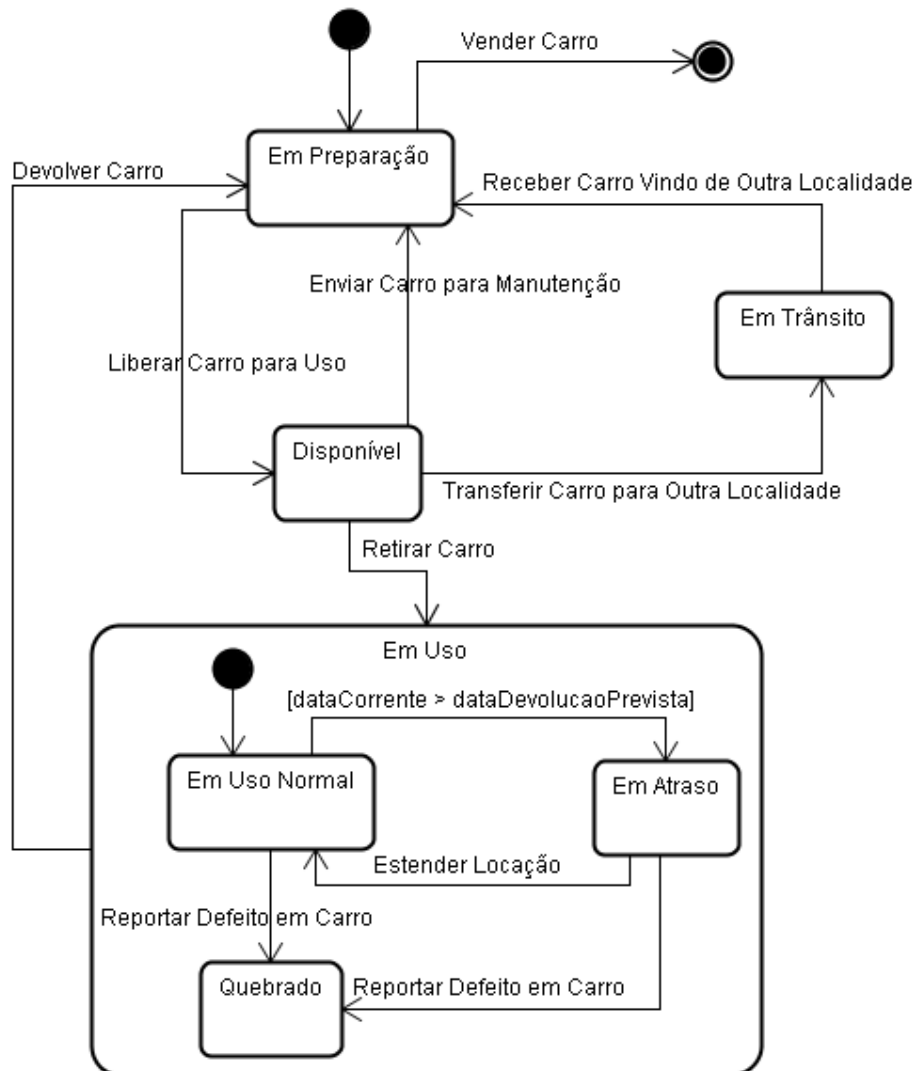
Figura 5.48 – Diagrama de Estados da Classe *Carro* (Disponibilidade) com Estado Composto *Em Uso* (adaptado de (OLIVÉ, 2007)).

O estado de um objeto deve ser mapeado no esquema estrutural. De maneira geral, o estado pode ser modelado por meio de um atributo. Esse atributo deve ser monovalorado e obrigatório ([1..1]). O conjunto de valores possíveis do atributo é o conjunto dos estados possíveis, conforme descrito pela máquina de estados (OLIVÉ, 2007). Assim, é bastante natural que o tipo de dados desse atributo seja definido como um tipo de dados enumerado. Um nome adequado para esse atributo é "estado". Contudo, outros nomes mais significativos para o domínio podem ser atribuídos. Em especial, quando uma classe possuir mais do que uma máquina de estado e, por conseguinte, mais do que um atributo de estado for necessário, o nome do atributo de estado deve indicar a perspectiva capturada pela correspondente máquina de estados.

É interessante observar que algumas transições podem mudar a estrutura da classe. Quando os diferentes estados de um objeto não afetam a sua estrutura, mas apenas, possivelmente, os valores de seus atributos e associações, diz-se que a transição é estável e os diferentes estados podem ser mapeados para um simples atributo (WAZLAWICK, 2004), conforme discutido anteriormente.

Entretanto, há situações em que, conforme um objeto vai passando de um estado para outro, ele vai ganhando novos atributos ou associações, ou seja, há uma mudança na estrutura da classe. Seja o exemplo de uma locação de carro. Como mostra a Figura 5.49, quando uma locação é criada, ela está ativa, em curso normal. Quando o carro não é devolvido até a data de devolução prevista, a locação passa a ativa com prazo expirado. Se a locação é estendida, ela volta a ficar em curso normal. Quando o carro é devolvido, a locação fica pendente. Finalmente, quando o pagamento é efetuado, a locação é concluída.
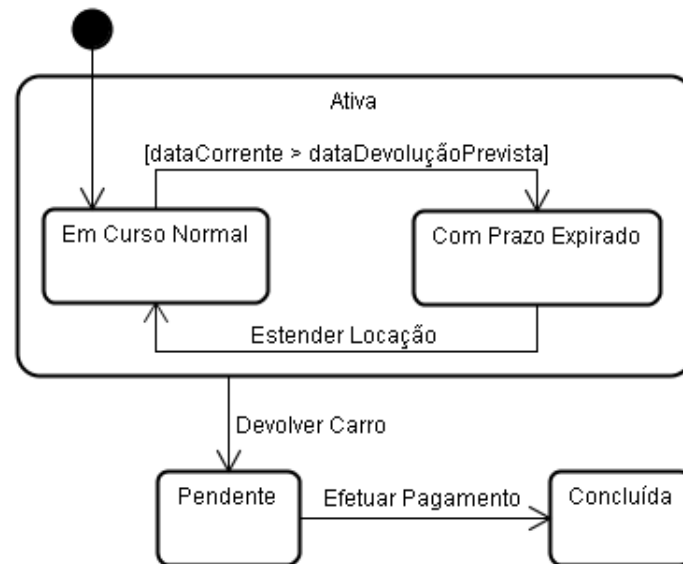
Figura 5.49 – Diagrama de Estados da Classe *Locação*.

Locações ativas (e em seus subestados, obviamente) têm como atributos: data de locação, data de devolução prevista, valor devido e caução. Quando no estado pendente, é necessário registrar a data de devolução efetiva e os problemas observados no carro devolvido. Finalmente, quando o pagamento é efetuado, é preciso registrar a data do pagamento, o valor e a forma de pagamento. Diz-se que as transições dos estados de *Ativa* para *Pendente* e de *Pendente* para *Concluída* são monotônicas[19], porque a cada mudança de estado, novos relacionamentos (atributos ou associações) são acrescentados (mas nenhum é retirado).

Uma solução frequentemente usada para capturar essa situação no modelo conceitual estrutural consiste em criar uma única classe (*Locacao*) e fazer com que certos atributos sejam nulos até que o objeto mude de estado, como ilustra a Figura 5.50. Essa forma de modelagem, contudo, pode não ser uma boa opção, uma vez que gera classes complexas com regras de consistência que têm de ser verificadas muitas vezes para evitar a execução de um método que atribui um valor a um atributo específico de um estado (WAZLAWICK, 2004), tal como *dataPagamento*.

[19] Monotônico diz respeito a algo que ocorre de maneira contínua. Neste caso, a continuidade advém do fato de um objeto continuamente ganhar novos atributos e associações, sem perder os que já possuía.

Figura 5.50 – Classe *Locação* com atributos inerentes a diferentes estados.

É possível modelar essa situação desdobrando o conceito original em três: um representando a locação efetivamente, outro representando a devolução e outro representando o pagamento. Desta forma, captura-se claramente os eventos de locação, devolução e pagamento, colocando as informações de cada evento na classe correspondente, como ilustra a Figura 5.51.
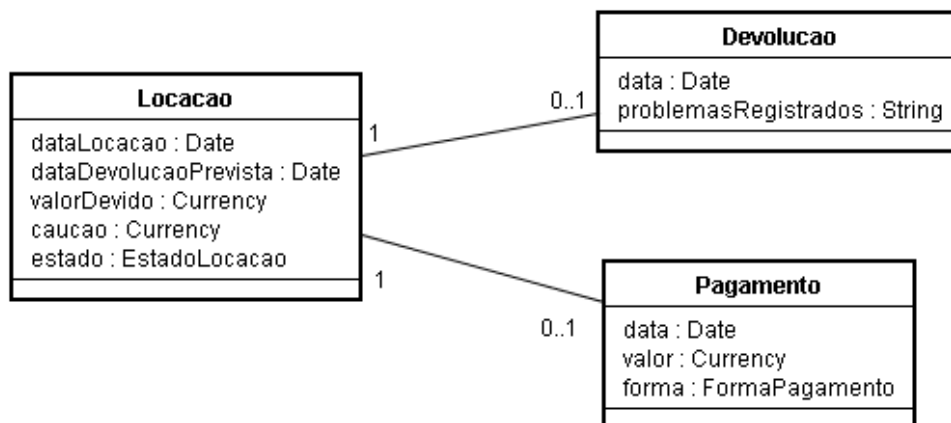
Figura 5.51– Distribuindo as responsabilidades.

Finalmente, vale à pena comentar que estados de uma classe modal podem ser tratados por meio de operações ao invés de atributos. Seja o exemplo anterior de locações de carros (Figura 5.51). O estado de uma locação pode ser computado a partir dos atributos e associações da classe *Locacao*, sem haver a necessidade de um atributo *estado*. Se uma locação não tem uma devolução associada, então ela está ativa. Estando ativa, se a data corrente é menor ou igual à data de devolução prevista, então a locação está em curso normal; caso contrário, ela está com prazo expirado. Se uma locação possui uma devolução, mas não possui um pagamento associado, então ela está pendente. Finalmente, se a locação possui um pagamento associado, então ela está concluída. Em casos como este, pode-se optar por tratar estado como uma operação e não como um atributo. Opcionalmente, pode-se utilizar a operação para calcular o valor de um atributo derivado[20] *estado*. Atributos derivados são representados na UML precedidos por uma barra (no exemplo, */estado*).

[20] Um atributo é derivado quando seu valor pode ser deduzido ou calculado a partir de outras informações (atributos e associações) já existentes no modelo estrutural.

Leitura Complementar

Engenharia de Requisitos

Em (SOMMERVILLE, 2007), a parte 2 – Requisitos – como o próprio nome indica, é dedicada ao tema. Merecem atenção especial os capítulos 6 e 7. O Capítulo 6 – Requisitos de Software – trata de tipos e níveis de requisitos, bem como da documentação de requisitos. O Capítulo 7 – Processos de Engenharia de Requisitos – apresenta e discute um processo de engenharia de requisitos (também muito similar ao apresentado neste texto) e suas atividades.

Em (PFLEEGER, 2004), o Capítulo 4 – Identificando Requisitos – tem uma boa discussão sobre requisitos. Em relação aos temas abordados nestas notas de aula, merecem destaque as seções 4.1, 4.2, 4.3, 4.6, 4.7, 4.8 e 4.9, as quais tratam dos seguintes assuntos: requisitos, processo de requisitos, tipos de documentos de requisitos, tipos de requisitos, características de requisitos (seções 4.1, 4.2 e 4.3), prototipagem de requisitos (seção 4.6), documentação de requisitos (seção 4.7), participantes no processo de requisitos (seção 4.8) e validação de requisitos (seção 4.9).

Em (PRESSMAN, 2006), o Capítulo 7 – Engenharia de Requisitos – aborda vários dos temas discutidos neste capítulo, com destaque para as seções 7.2 (Tarefas da Engenharia de Requisitos), 7.3 (Início do Processo de Engenharia de Requisitos), 7.4 (Levantamento de Requisitos), 7.7 (Negociação de Requisitos) e 7.8 (Validação de Requisitos).

Modelagem de Casos de Uso

Os capítulos 7 e 8 de (BLAHA; RUMBAUGH, 2006) – *Modelagem de Interações* e *Modelagem Avançada de Interações*, respectivamente – abordam a modelagem de casos de uso. Mais especificamente, recomenda-se a leitura da seção 7.1 (*Modelos de Casos de Uso*), que dá uma visão geral de atores, casos de uso e diagramas de casos de uso, e da seção 8.1 (*Relações entre Casos de Uso*), que discute as relações de inclusão, extensão e generalização e especialização entre casos de uso.

O Capítulo 15 de (OLIVÉ, 2007) – *Use Cases* – dá uma visão geral da modelagem de casos de uso, discutindo de maneira breve, mas bastante didática, os conceitos de ator e de caso de uso, a especificação de casos de uso e os relacionamentos entre casos de uso.

O livro "*Escrevendo Casos de Uso Eficazes: Um guia prático para desenvolvedores de software*" (COCKBURN, 2005) é inteiramente dedicado ao processo de escrita de casos de uso. Esse livro é uma ótima referência para os interessados em aperfeiçoar seu processo de escrita de casos de uso, contendo diversas diretrizes incorporadas nestas notas de aula.

Em (WAZLAWICK, 2004), tanto o Capítulo 2 (*Concepção*) quanto o Capítulo 3 (*Expansão dos Casos de Uso*) abordam a modelagem de casos de uso.

Em (BOOCH; RUMBAUGH; JACOBSON, 2006), merecem atenção os capítulos 17 (*Casos de Uso*) e 18 (*Diagramas de Casos de Uso*). As notações da UML para diagramas de casos de uso são tratadas com mais detalhes do que nas demais referências citadas anteriormente, precisamente por se tratar este de um livro sobre a UML.

Modelagem Conceitual Estrutural

O Capítulo 5 de (WAZLAWICK, 2004) – *Modelagem Conceitual* – dá uma visão geral da modelagem estrutural, discutindo de maneira bastante didática, diversos de seus aspectos.

Em (BOOCH; RUMBAUGH; JACOBSON, 2006), merecem atenção os capítulos 4 (*Classes*), 5 (*Relacionamentos*), 8 (*Diagramas de Classes*), 9 (*Classes Avançadas*) e 10 (*Relacionamentos Avançados*). As notações da UML para diagramas de classes são tratadas com mais detalhes do que nas demais referências citadas anteriormente, precisamente por se tratar este de um livro sobre a UML.

Diagrama de Estados

Em (BOOCH; RUMBAUGH; JACOBSON, 2006), merecem atenção os capítulos 22 (*Máquinas de Estados*) e 25 (*Diagramas de Gráficos de Estados*). As notações da UML para os diagramas abordados neste capítulo são tratadas com bastante detalhes, uma vez que este é um livro sobre a UML. Além desses capítulos, merece atenção a parte do Capítulo 9 (Classes Avançadas) que trata da sintaxe de operações.

Finalmente, em (BLAHA; RUMBAUGH, 2006), os capítulos 5, 6 e 12 abordam o desenvolvimento de diagramas de estados.

Referências do Capítulo

BLAHA, M., RUMBAUGH, J., *Modelagem e Projetos Baseados em Objetos com UML 2*, Elsevier, 2006.

BOOCH, G., *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings Publishing Company, Inc, 1994.

BOOCH, G., RUMBAUGH, J., JACOBSON, I., *UML Guia do Usuário*, 2a edição, Elsevier Editora, 2006.

COCKBURN, A., *Escrevendo Casos de Uso Eficazes: Um guia prático para desenvolvedores de software*, Porto Alegre: Bookman, 2005.

OLIVÉ, A., *Conceptual Modeling of Information Systems*, Springer, 2007.

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.

PRESSMAN, R.S., *Engenharia de Software*, McGraw-Hill, 6ª edição, 2006.

SOMMERVILLE, I., *Engenharia de Software*, 8ª Edição. São Paulo: Pearson – Addison Wesley, 2007.

TOGNERI, D.F., Apoio Automatizado à Engenharia de Requisitos Cooperativa. Dissertação (Mestrado em Informática), Universidade Federal do Espírito Santo (UFES), Vitória, 2002.

WAZLAWICK, R.S., *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, Elsevier, 2004.

WIEGERS, K.E., *Software Requirements: Practical techniques for gathering and managing requirements throughout the product development cycle*. 2nd Edition, Microsoft Press, Redmond, Washington, 2003.

YOURDON, E., *Object-Oriented Systems Design: an Integrated Approach*, Yourdon Press Computing Series, Prentice Hall, 1994.

Capítulo 6 – Projeto de Sistemas

O objetivo da fase de projeto (ou design) é produzir uma solução para o problema identificado e modelado na fase de análise, incorporando a tecnologia aos requisitos essenciais do usuário e projetando o que será construído na implementação. Sendo assim, é necessário conhecer a tecnologia disponível e os ambientes de software onde o sistema será desenvolvido e implantado. Durante o projeto, deve-se decidir como o problema será resolvido, começando em um alto nível de abstração, próximo da análise, detalhando depois até um nível de abstração próximo da implementação.

O projeto de software encontra-se no núcleo técnico do processo de desenvolvimento de software e é aplicado independentemente do modelo de ciclo de vida e paradigma adotados. É iniciado assim que os requisitos do software tiverem sido modelados e especificados pelo menos parcialmente e é a última atividade de modelagem. Por outro lado, corresponde à primeira atividade que leva em conta considerações de caráter tecnológico (PRESSMAN, 2006).

Enquanto a fase de análise pressupõe que a tecnologia é perfeita (capacidade ilimitada de processamento com velocidade instantânea, capacidade ilimitada de armazenamento, custo zero e não passível de falha), a fase de projeto envolve a modelagem de como o sistema será implementado com a adição dos requisitos tecnológicos ou não funcionais. Assim, como bem disse Mitch Kapor, citado por Pressman (2006), o projeto é "onde você se instala com um pé em dois mundos – o mundo da tecnologia e o mundo das pessoas e objetivos humanos – e você tenta juntar os dois". A Figura 6.1 procura ilustrar essa situação. O projeto é, portanto, a fase do processo de software na qual os requisitos do cliente, as necessidades do negócio e as considerações técnicas se juntam na formulação de um produto ou sistema (PRESSMAN, 2006).
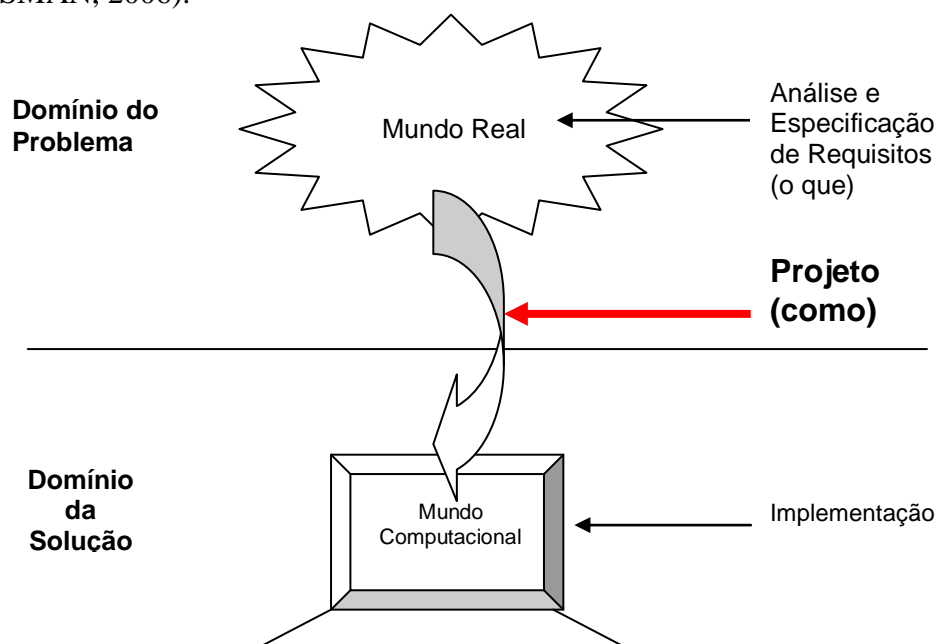
Figura 6.1 – Visão geral da fase de Projeto.

O projeto (design) é um processo de refinamento. Inicialmente, o projeto é representado em um nível alto de abstração, enfocando a estrutura geral do sistema. Definida a arquitetura, o projeto passa a tratar do detalhamento de seus elementos. Esses refinamentos conduzem a representações de menores níveis de abstração, até se chegar ao projeto de algoritmos e estruturas de dados. Assim, independentemente do paradigma adotado, o processo de projeto envolve as seguintes atividades:

- Projeto da Arquitetura do Software: visa definir os elementos estruturais do software e seus relacionamentos.
- Projeto dos Elementos da Arquitetura: visa projetar em um maior nível de detalhes cada um dos elementos estruturais definidos na arquitetura, o que envolve a decomposição de módulos em outros módulos menores.
- Projeto de Interfaces: tem por objetivo descrever como deverá se dar a comunicação entre os elementos da arquitetura (interfaces internas), a comunicação do sistema em desenvolvimento com outros sistemas (interfaces externas) e com as pessoas que vão utilizá-lo (interface com o usuário).
- Projeto Detalhado: visa refinar e detalhar a descrição procedimental e das estruturas de dados dos elementos mais básicos da arquitetura do software.

Tendo em vista que a orientação a objetos é um dos paradigmas mais utilizados atualmente no desenvolvimento de sistemas, este texto aborda o projeto de software orientado a objetos. Além disso, o foco deste texto são os sistemas de informação. Considerando essa classe de sistemas, de maneira geral, os seguintes elementos estão presentes na arquitetura de um sistema:

- Lógica de Domínio: é o elemento da arquitetura que trata de toda a lógica do sistema, englobando tanto aspectos estruturais (classes de domínio derivadas dos modelos conceituais estruturais da fase de análise), quanto comportamentais (classes de processo que tratam das funcionalidades descritas pelos casos de uso).
- Interface com o Usuário: é o elemento da arquitetura que trata da interação humano-computador. Envolve tanto as interfaces propriamente ditas (objetos gráficos responsáveis por receber dados e comandos do usuário e apresentar resultados) quanto o controle da interação, abrindo e fechando janelas, habilitando ou desabilitando botões etc (WAZLAWICK, 2004).
- *Persistência*: é o elemento da arquitetura responsável pelo armazenamento e recuperação de dados em memória secundária (classes que representam e isolam os depósitos de dados do restante do sistema).

6.1 Aspectos Relevantes ao Projeto de Software

Nesta seção são discutidos alguns aspectos relevantes quando se fala em Projeto de Software: qualidade, arquitetura, padrões e documentação.

6.1.1 Qualidade do Projeto de Software

Um bom projeto de software deve apresentar determinadas características de qualidade, tais como facilidade de entendimento, facilidade de implementação, facilidade de realização de testes, facilidade de modificação e tradução correta das especificações de requisitos e de análise (PFLEEGER, 2004). Para se obter bons projetos, é necessário considerar alguns aspectos intimamente relacionados com a qualidade dos projetos, dentre eles (PRESSMAN, 2006):

- **Níveis de Abstração**: a abstração é um dos modos fundamentais pelos quais os seres humanos enfrentam a complexidade. Assim, um bom projeto deve considerar vários níveis de abstração, começando com em um nível mais alto, próximo da fase de análise. À medida que se avança no processo de projeto, o nível de abstração deve ser reduzido. Dito de outra maneira, o projeto deve ser um processo de refinamento, no qual o projeto vai sendo conduzido de níveis mais altos para níveis mais baixos de abstração.

- **Modularidade**: um bom projeto deve estruturar um sistema como módulos ou componentes coesos e fracamente acoplados. A modularidade é o atributo individual que permite a um projeto de sistema ser intelectualmente gerenciável. A estratégia "dividir para conquistar" é reconhecidamente útil no projeto de software, pois é mais fácil resolver um problema complexo quando o mesmo é dividido em partes menores gerenciáveis

- **Ocultação de Informações**: o conceito de modularidade leva o projetista a uma questão fundamental: até que nível a decomposição deve ser aplicada? Em outras palavras, quão modular deve ser o software? O princípio da ocultação de informações sugere que os módulos / componentes sejam caracterizados pelas decisões de projeto que cada um deles esconde dos demais. Módulos devem ser projetados e especificados de modo que as informações neles contidas (dados e algoritmos) sejam inacessíveis a outros módulos, sendo necessário conhecer apenas a sua interface. Ou seja, a ocultação de informação trabalha encapsulando detalhes que provavelmente serão alterados independentemente em diferentes módulos. A interface de um módulo revela apenas aqueles aspectos considerados improváveis de mudar (BASS; CLEMENTS; KAZMAN, 2003).

- **Independência Funcional**: a independência funcional é uma decorrência direta da modularidade e dos conceitos de abstração e ocultação de informações. Ela é obtida pelo desenvolvimento de módulos com finalidade única e pequena interação com outros módulos, isto é, módulos devem cumprir uma função bem estabelecida, minimizando interações com outros módulos. Módulos funcionalmente independentes são mais fáceis de entender, desenvolver, testar e alterar. Efeitos colaterais causados pela modificação de um módulo são limitados e, por conseguinte, a propagação de erros é reduzida. A independência funcional pode ser avaliada usando dois critérios de qualidade: coesão e acoplamento. A coesão se refere ao elo de ligação com o qual um módulo é construído. Uma classe, p.ex., é dita coesa quando tem um conjunto pequeno e focado de responsabilidades e aplica seus atributos e métodos especificamente para implementar essas responsabilidades. Já o acoplamento diz respeito ao grau de interdependência entre dois módulos. O objetivo é minimizar o acoplamento, isto é, tornar os módulos tão independentes quanto possível.

Idealmente, classes de projeto em um subsistema deveriam ter conhecimento limitado de classes de outros subsistemas. Coesão e acoplamento são interdependentes e, portanto, uma boa coesão deve conduzir a um pequeno acoplamento. A Figura 6.2 procura ilustrar esse fato.
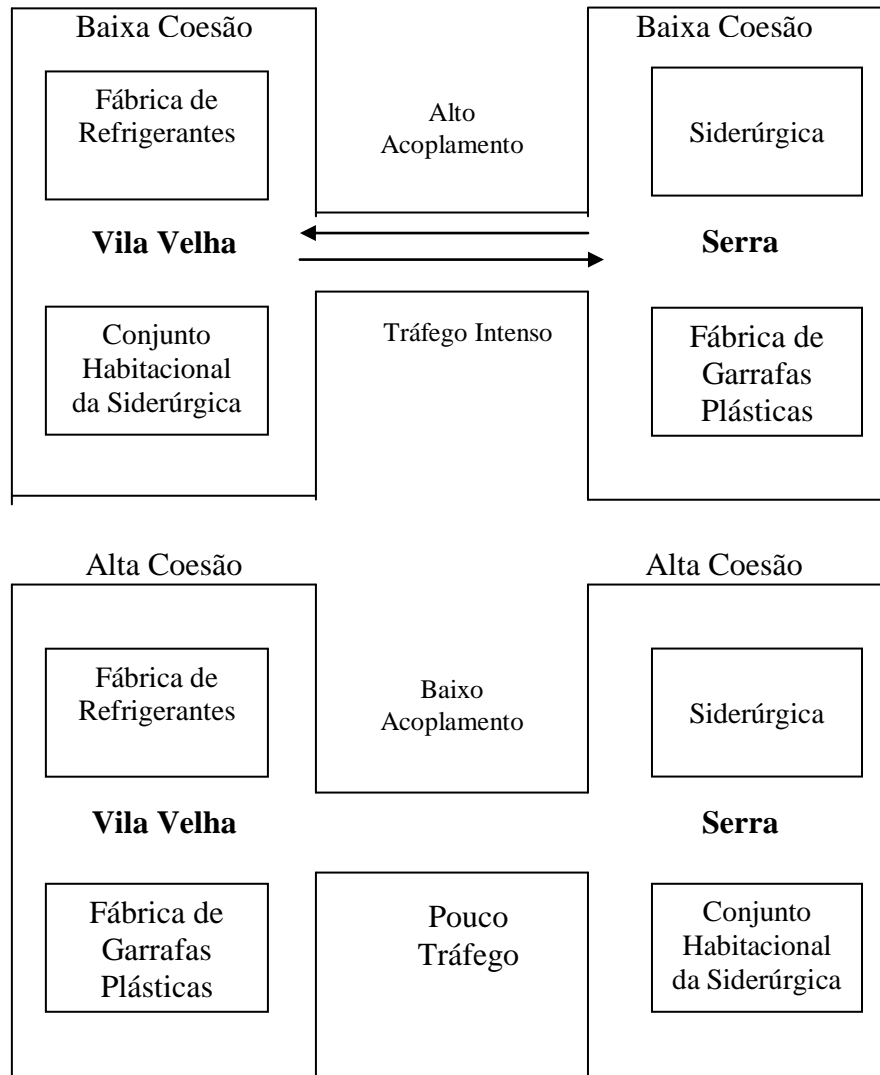
Figura 6.2 – Coesão e Acoplamento.

6.1.2 Arquitetura de Software

De acordo com Bass, Clements e Kazman (2003), a arquitetura de software de um sistema computacional é a estrutura (ou estruturas) do sistema que compreende elementos de software, propriedades externamente visíveis desses elementos e os relacionamentos entre eles. A arquitetura define elementos de software, ou módulos, e envolve informação sobre como eles se relacionam uns com os outros. Uma arquitetura pode envolver mais de um tipo de estrutura, com diferentes tipos de elementos e de relacionamentos entre elementos. A arquitetura omite certas informações sobre os elementos que não pertencem às suas interações. As propriedades externamente visíveis indicam as suposições que os demais elementos podem fazer sobre um elemento, tais como serviços providos e características de qualidade esperadas. Assim, uma arquitetura

é antes de tudo uma abstração de um sistema que suprime detalhes dos elementos que não afetam como eles são usados, como se relacionam, como interagem e como usam outros elementos. Na maioria das vezes, a arquitetura é usada para descrever aspectos estruturais de um sistema.

Em quase todos os sistemas modernos, elementos interagem com outros por meio de interfaces que dividem detalhes sobre um elemento em partes pública e privada. A arquitetura está preocupada com a parte pública dessa divisão. Detalhes privados, aqueles que têm a ver somente com a implementação interna, não são arquiteturais (BASS; CLEMENTS; KAZMAN, 2003).

Até o projeto arquitetônico, aspectos relacionados ao hardware e à plataforma de implementação ainda não foram tratados, já que a fase de análise pressupõe tecnologia perfeita. Este é o momento para resolver como um modelo ideal vai executar em uma plataforma restrita. É importante realçar que não existe a solução perfeita. O projeto da arquitetura é uma tarefa de negociação, onde se faz compromissos entre soluções sub-ótimas. O modelo de arquitetura mapeia os requisitos essenciais da fase de análise em uma arquitetura técnica. Uma vez que muitas arquiteturas diferentes são possíveis, o propósito do projeto arquitetônico é escolher a configuração mais adequada. Além disso, fatores que transcendem aspectos puramente técnicos devem ser considerados.

Normalmente, o projeto da arquitetura é discutido à luz dos requisitos do sistema, ou seja, se os requisitos são conhecidos, então se pode projetar a arquitetura do sistema. Contudo, deve-se considerar o projeto arquitetônico como algo mais abrangente, envolvendo aspectos técnicos, sociais e de negócio. Todos esses fatores (e não somente os requisitos do sistema) influenciam a arquitetura de software. Esta, por sua vez, afeta o ambiente da organização (incluindo ambientes técnico, social e de negócio) que vai influenciar arquiteturas futuras, criando um ciclo de realimentação contínua. Por exemplo, se os projetistas encarregados do projeto de um novo sistema obtiveram bons resultados em projetos de sistemas anteriores usando uma particular abordagem de arquitetura, então é natural que eles tentem a mesma abordagem no novo projeto. Por outro lado, se suas experiências anteriores com essa abordagem foram desastrosas, os projetistas vão relutar em tentá-la outra vez, mesmo que ela se apresente como uma solução adequada. Assim, as escolhas são guiadas, também, pela formação e experiência dos projetistas (BASS; CLEMENTS; KAZMAN, 2003).

Outro fator que afeta a escolha da arquitetura é o ambiente técnico (ou plataforma de implementação) corrente. Muitas vezes, há para esse ambiente um conjunto dominante de padrões, práticas e técnicas que é aceito pela comunidade de arquitetos ou pela organização de desenvolvimento. Por fim, a arquitetura é influenciada também pela estrutura e natureza da organização de desenvolvimento (BASS; CLEMENTS; KAZMAN, 2003).

Assim, no projeto da arquitetura de software, projetistas são influenciados por requisitos para o sistema, estrutura e metas da organização de desenvolvimento, ambiente técnico disponível e por suas próprias experiências e formação. Além disso, os relacionamentos entre metas de negócio, requisitos de sistemas, experiência dos projetistas, arquiteturas e sistemas implantados geram diversos laços de realimentação que podem ser gerenciado pela organização.

Muitas pessoas têm interesse na arquitetura de software, tais como clientes, usuários finais, desenvolvedores, gerentes de projeto e mantenedores. Alguns desses interesses são conflitantes e o projetista frequentemente tem de mediar conflitos até

chegar à configuração que atenda de forma mais adequada a todos os interesses. Neste contexto, arquiteturas de software são importantes principalmente porque (BASS; CLEMENTS; KAZMAN, 2003):

- Representam uma abstração comum do sistema que pode ser usada para compreensão mútua, negociação, consenso e comunicação entre os interessados. A arquitetura provê uma linguagem comum na qual diferentes preocupações podem ser expressas, negociadas e resolvidas em um nível que seja intelectualmente gerenciável.
- Manifestam as primeiras decisões de projeto. Essas decisões definem restrições sobre a implementação e a estrutura organizacional do projeto. A implementação tem de ser dividida nos elementos prescritos pela arquitetura. Os elementos têm de interagir conforme o prescrito e cada elemento tem de cumprir sua responsabilidade conforme ditado pela arquitetura. Também a estrutura organizacional do projeto, e às vezes até a estrutura da organização como um todo, torna-se amarrada à estrutura proposta pela arquitetura. Neste sentido, a arquitetura pode ajudar a obter estimativas e cronogramas mais precisos, bem como pode ajudar na prototipagem do sistema. Além disso, a extensão na qual o sistema vai ser capaz de satisfazer os atributos de qualidade requeridos é substancialmente determinada pela arquitetura. Particularmente a manutenibilidade é fortemente afetada pela arquitetura. Uma arquitetura reparte possíveis alterações em três categorias: locais (confinadas em um único elemento), não locais (requerem a alteração de vários elementos, mas mantêm intacta a abordagem arquitetônica subjacente) e arquitetônicas (afetam a estrutura do sistema e podem requerer alterações ao longo de todo o sistema). Obviamente, alterações locais são as mais desejáveis e, portanto, uma arquitetura efetiva deve propiciar que as alterações mais prováveis sejam as mais fáceis de fazer.
- Constituem um modelo relativamente pequeno e intelectualmente compreensível de como o sistema é estruturado e como seus elementos trabalham em conjunto. Além disso, esse modelo é transferível para outros sistemas, em especial para aqueles que exibem requisitos funcionais e não funcionais similares, promovendo reúso em larga escala. Um desenvolvimento baseado na arquitetura frenquentemente enfoca a composição ou montagem de elementos que provavelmente foram desenvolvidos separadamente, até mesmo de forma independente. Essa composição é possível porque a arquitetura define os elementos que devem ser incorporados no sistema. Além disso, a arquitetura restringe possíveis substituições de elementos segundo a forma como eles interagem com o ambiente, recebem e entregam o controle, que dados consomem e produzem, como acessam esses dados e quais protocolos usam para se comunicar e compartilhar recursos.

É importante que o projetista seja capaz de reconhecer estruturas comuns utilizadas em sistemas já desenvolvidos, de modo a poder compreender as relações existentes e desenvolver novos sistemas como variações dos sistemas existentes. O entendimento de arquiteturas de software existentes permite que os projetistas avaliem alternativas de projeto. Neste contexto, uma representação da arquitetura é essencial para permitir descrever propriedades de um sistema complexo, bem como uma análise da arquitetura proposta (MENDES, 2002).

Muitas vezes, arquiteturas são representadas na forma de diagramas contendo caixas (representando elementos) e linhas (representando relacionamentos). Entretanto, tais diagramas não dizem muita coisa sobre o que são os elementos e como eles cooperam para realizar o propósito do sistema e, portanto, não capturam importantes informações de arquitetura (BASS; CLEMENTS; KAZMAN, 2003). Por exemplo, em uma visão de decomposição de módulos, é importante distinguir quando um módulo é decomposto em outros módulos e quando um módulo simplesmente usa outros módulos. Já em uma arquitetura cliente-servidor, é importante apontar quando um módulo é considerado cliente e quando ele é considerado servidor. Assim, idealmente, a representação de uma arquitetura deve incorporar informações acerca dos tipos dos elementos e dos relacionamentos.

6.1.3 Padrões *(Patterns)*

Todo projeto de desenvolvimento é, de alguma maneira, novo, na medida em que se quer desenvolver um novo sistema, seja porque ainda não existe um sistema para resolver o problema que está sendo tratado, seja porque há aspectos indesejáveis nos sistemas existentes. Isso não quer dizer que o projeto tenha que ser desenvolvido a partir do zero. Muito pelo contrário. A reutilização é um aspecto fundamental no desenvolvimento de software. Muitos sistemas previamente desenvolvidos são similares ao sistema em desenvolvimento e há muito conhecimento que pode ser reaplicado para solucionar questões recorrentes no projeto de software. Os padrões (*patterns*) visam capturar esse conhecimento, procurando torná-lo mais geral e amplamente aplicável, desvinculando-o das especificidades de um determinado projeto ou sistema.

Um padrão é uma solução testada e aprovada para um problema geral. Diferentes padrões se destinam a diferentes fases do ciclo de vida: análise, arquitetura, projeto e implementação. Um padrão vem com diretrizes sobre quando usá-lo, bem como vantagens e desvantagens de seu uso. Um padrão já foi cuidadosamente considerado por outras pessoas e aplicado diversas vezes na solução de problemas anteriores de mesma natureza. Assim, tende a ser uma solução de qualidade, com maiores chances de estar correto e estável do que uma solução nova, específica, ainda não testada (BLAHA; RUMBAUGH, 2006).

Um padrão normalmente tem o formato de um par nomeado problema/solução, que pode ser utilizado em novos contextos, com orientações sobre com utilizá-lo nessas novas situações (LARMAN, 2007). O objetivo de um padrão de projeto é registrar uma experiência no projeto de software, que possa ser efetivamente utilizado por projetistas. Cada padrão sistematicamente nomeia, explica e avalia uma importante situação de projeto que ocorre repetidamente em sistemas (GAMMA et al., 1995).

Um projetista familiarizado com padrões pode aplicá-los diretamente a problemas sem ter que redescobrir as abstrações e os objetos que as capturam. Uma vez que um padrão é aplicado, muitas decisões de projeto decorrem automaticamente.

Em geral, um padrão tem os seguintes elementos (GAMMA et al., 1995) (BUSCHMANN et al., 1996):

- **Nome**: identificação de uma ou duas palavras, utilizada para nomear o padrão.
- **Contexto**: uma situação que dá origem a um problema.

- **Problema**: explica o problema que surge repetidamente no dado contexto.
- **Solução**: descreve uma solução comprovada para o problema, incluindo os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. É importante observar que não descreve um particular projeto concreto ou implementação. Um padrão provê uma descrição abstrata de um problema de projeto e como uma organização geral de elementos resolve esse problema.
- **Consequências**: são os resultados e os comprometimentos feitos ao se aplicar o padrão.

No que concerne aos padrões relacionados à fase de projeto, há três grandes categorias a serem consideradas:

- Padrões Arquitetônicos: definem uma estrutura global do sistema. Um padrão arquitetônico indica um conjunto predefinido de subsistemas, especifica as suas responsabilidades e inclui regras e orientações para estabelecer os relacionamentos entre eles. São aplicados na atividade de projeto da arquitetura de software e podem ser vistos como modelos (*templates*) para arquiteturas de software concretas (BUSCHMANN et al., 1996).
- Padrões de Projeto (*Design Patterns*): atendem a uma situação específica de projeto, mostrando classes e relacionamentos, seus papéis e suas colaborações e também a distribuição de responsabilidades. Um padrão de projeto provê um esquema para refinar subsistemas ou componentes de sistema de software, ou os relacionamentos entre eles. Ele descreve uma estrutura comumente recorrente de componentes que se comunicam, a qual resolve um problema de projeto geral dentro de um particular contexto (GAMMA et al., 1995).
- Idiomas: representam o nível mais baixo de padrões, endereçando aspectos tanto de projeto quanto de implementação. Um idioma é um padrão de baixo nível, específico de uma linguagem de programação, descrevendo como implementar aspectos particulares de componentes ou os relacionamentos entre eles usando as características de uma dada linguagem (BUSCHMANN et al., 1996).

6.1.4 Documentação de Projeto

Uma vez que o projeto de software encontra-se no núcleo técnico do processo de desenvolvimento, sua documentação tem grande importância para o sucesso do projeto e para a manutenção futura do sistema. Diferentes interessados vão requerer informações diferentes e a documentação de projeto é crucial para a comunicação. Analistas, projetistas e clientes vão precisar negociar para estabelecer prioridades entre requisitos conflitantes; programadores e testadores vão utilizar essa documentação para implementar e testar o sistema; gerentes de projeto vão usar informações da decomposição do sistema para definir e alocar equipes de trabalho; mantenedores vão recorrer a essa documentação na hora de avaliar e realizar uma alteração.

Uma vez que o projeto é um processo de refinamento, a sua documentação também deve prover representações em diferentes níveis de abstração. Além disso, o

projeto de um sistema é uma entidade complexa que não pode ser descrita em uma única perspectiva. Ao contrário, múltiplas visões são essenciais e a documentação deve abranger aquelas visões consideradas relevantes. De fato, como muitas visões são possíveis, a documentação é uma atividade que envolve a escolha das visões relevantes, a documentação das visões selecionadas e a documentação de informações que se aplicam a mais do que uma visão (BASS; CLEMENTS; KAZMAN, 2003). A escolha das visões é dependente de vários fatores, dentre eles, do tipo de sistema sendo desenvolvido, dos atributos de qualidade considerados e da audiência da documentação de projeto. Diferentes visões realçam diferentes elementos de um sistema.

6.2 Projetando a Arquitetura de Software

Projetar a arquitetura de um software requer o levantamento de informações relativas à plataforma de computação do sistema, as quais se somarão ao conhecimento acerca dos requisitos funcionais e não funcionais, para dar embasar as decisões relativas à arquitetura do sistema que está sendo projetado. De maneira geral, o processo de projetar envolve, dentre outros, os seguintes passos:

1. Levantar informações acerca da plataforma de computação do sistema, incluindo linguagem de programação a ser adotada, mecanismo de persistência e necessidades de distribuição geográfica.
2. Com base nos requisitos, iniciar a decomposição do sistema em subsistemas, considerando preferencialmente uma decomposição pelo domínio do problema. Se na fase de análise já tiver sido estabelecida uma decomposição inicial em subsistemas, esta deverá ser utilizada. Neste momento, deve-se escolher um estilo arquitetônico (ou uma combinação adequada de estilos arquitetônicos) para organizar a estrutura geral do sistema.
3. Estabelecer uma arquitetura base, identificando tipos de módulos e tipos de relacionamentos entre eles, dados essencialmente pela combinação de estilos arquitetônicos escolhidos.
4. Alocar requisitos funcionais (casos de uso) e não funcionais aos componentes da arquitetura.
5. Avaliar a arquitetura, procurando identificar se ela acomoda os requisitos e restrições identificados.
6. Uma vez definida a arquitetura em seu nível mais alto, passar ao projeto de seus elementos. Padrões arquitetônicos são instrumentos muito valiosos para auxiliar o projeto dos componentes da arquitetura.

Em relação ao passo 2, um estilo arquitetônico que combina partições e camadas tende a ser um bom ponto de partida para a estruturação global de sistemas de informação.

- **Partições** podem ser derivadas a partir do domínio do problema, levando-se em conta funcionalidades coesas, visando à criação de subsistemas fracamente acoplados. Idealmente, alguma divisão em subsistemas deve ter sido feita na fase de análise e a mesma deve ser aqui preservada.
- As partições podem ser estruturadas em camadas e novamente um bom ponto de partida é considerar **camadas** típicas de um sistema de informação

(Interface com o Usuário, Domínio do Problema e Gerência de Dados), mostradas na Figura 6.3.
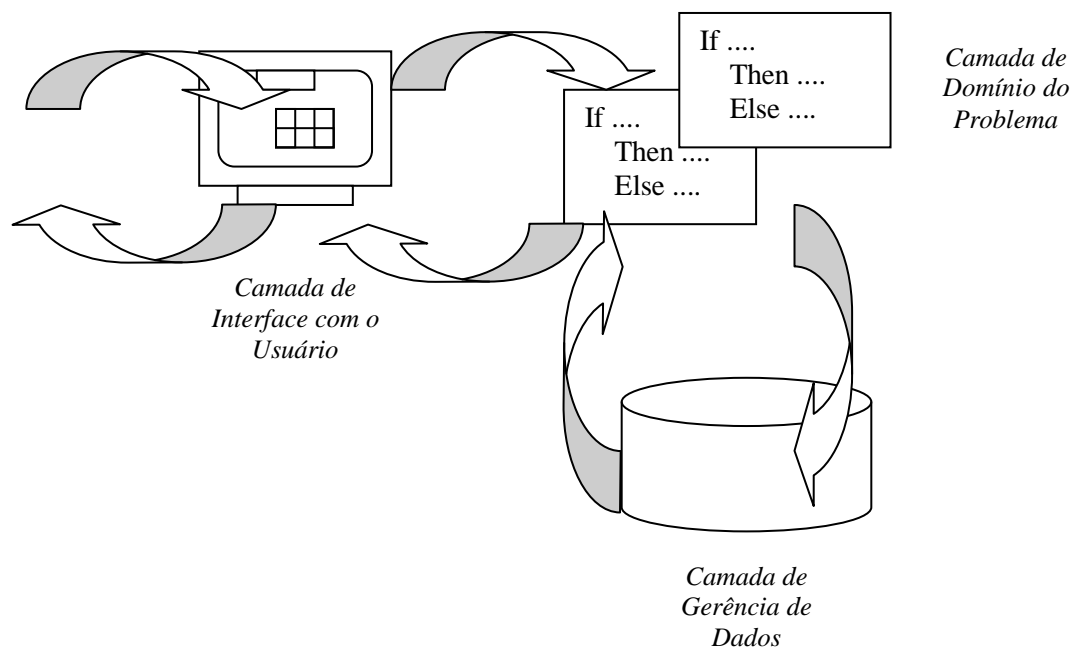
Figura 6.3 – Camadas típicas em sistemas de informação.

Nas próximas seções são apresentadas discussões relacionadas ao projeto de cada uma dessas camadas.

6.3 A Camada de Domínio do Problema

A camada de lógica de negócio engloba o conjunto de classes que vai realizar toda a lógica do sistema de informação. As demais camadas são derivadas ou dependentes da camada de domínio (WAZLAWICK, 2004) e, portanto, é interessante iniciar o projeto dos componentes da arquitetura do sistema pela camada da lógica de negócio.

Os modelos construídos na fase de análise são os principais insumos para o projeto dessa camada, em especial o modelo conceitual e o modelo de casos de uso. Os diagramas de classes da fase de análise serão a base para a construção dos diagramas de classes da fase de projeto. De fato, a versão inicial do modelo estrutural de projeto (diagramas de classes de projeto) da lógica de negócio será uma cópia do modelo conceitual estrutural. Durante o projeto, esse modelo será objeto de refinamento, visando incorporar informações importantes para a implementação, tais como distribuição de responsabilidades entre as classes (definição de métodos das classes) e definição de navegabilidades, visibilidades e tipos de dados. Além disso, alterações na estrutura do diagrama de classes podem ser necessárias para tratar requisitos não funcionais, tais como usabilidade e desempenho.

Para organizar a lógica de negócio, um bom ponto de partida são os padrões arquitetônicos relativos a essa camada. No contexto do desenvolvimento de Sistemas de Informação Orientados a Objetos, merecem destaque os padrões **Modelo de Domínio** e **Camada de Serviço**.

Uma questão bastante importante tratada por esses padrões é a distribuição de responsabilidades ao longo das classes que compõem o sistema. No mundo de objetos, uma funcionalidade é realizada através de uma rede de objetos interconectados, colaborando entre si. Objetos encapsulam dados e comportamento. O posicionamento correto do comportamento na rede de objetos é um dos principais problemas a serem enfrentados durante o projeto da lógica de negócio. Neste contexto, um desafio a mais se coloca: que classes vão comportar as funcionalidades descritas pelos casos de uso? Duas formas básicas são comumente adotadas:

- Distribuir as responsabilidades para a execução dos casos de uso ao longo dos objetos do domínio do problema: essa abordagem é refletida no padrão Modelo de Domínio e considera que as funcionalidades relativas aos casos de uso do sistema estarão distribuídas nas classes previamente identificadas na fase de análise (Componente Domínio do Problema).
- Considerar que a lógica de negócio é, na verdade, composta por dois tipos de lógica: a lógica de domínio do problema (Componente de Domínio do Problema), que tem a ver puramente com as classes previamente identificadas na fase de análise; e lógica da aplicação (Componente de Gerência de Tarefas), que se refere às funcionalidades descritas pelos casos de uso. Esta segunda opção é a essência do padrão Camada de Serviço.

A seguir são apresentados os padrões Modelo de Domínio e Camada de Serviço. Depois, são apresentados o Componente Domínio do Problema, necessário tanto para o padrão Modelo de Domínio quanto para o padrão Camada de Serviço, e o Componente de Gerência de Tarefas, necessário apenas no padrão Camada de Serviço.

6.3.1 Padrões Arquitetônicos para o Projeto da Lógica de Negócio

Um importante aspecto do projeto da Camada de Lógica de Negócio diz respeito à organização das classes e distribuição de responsabilidades entre elas, o que vai definir, em última instância, os métodos de cada classe dessa camada. Nesse contexto, dpos padrões arquitetônicos merecem destaque: Modelo de Domínio e Camada de Serviço.

De forma resumida, no padrão Modelo de Domínio, as responsabilidades são distribuídas nos objetos do domínio do problema e a lógica de aplicação é pulverizada nesses objetos, sendo que cada objeto tem uma parte da lógica que é relevante a ele. Já na abordagem do padrão Camada de Serviço, um conjunto de objetos controladores de casos de uso[21] fica responsável por tratar a lógica de aplicação, controlando o fluxo de eventos dentro do caso de uso. Grande parte da lógica de negócio ainda fica a cargo dos objetos do domínio do problema. Cabe aos controladores de casos de uso apenas centralizar o controle sobre a execução do caso de uso. Assim, a camada de serviço é construída, de fato, sobre a camada de domínio.

A seguir os padrões são apresentados em mais detalhes.

Padrão Modelo de Domínio

O padrão Modelo de Domínio preconiza que o próprio modelo de objetos do domínio incorpore dados e comportamento. Um modelo de domínio estabelece uma

[21] Classes controladoras de caso de uso são classes que centralizam as interações no contexto de casos de uso específicos e não são consideradas controladores no sentido usado no padrão MVC.

rede de objetos interconectados, onde cada objeto representa alguma entidade significativa no mundo real. Colocar um modelo de domínio em uma aplicação envolve inserir uma camada de objetos que modela a área de negócio da aplicação. Os objetos dessa camada vão ser abstrações de entidades do negócio que capturam regras que o negócio utiliza. Os dados e os processos são combinados para agrupar os processos próximos dos dados com os quais eles trabalham (FOWLER, 2003). Neste sentido, pode-se dizer que o padrão Modelo de Domínio captura a essência da orientação a objetos. Como benefícios, têm-se os benefícios propalados pela orientação a objetos, tais como evitar duplicação da lógica de negócio e gerenciar a complexidade usando *design patterns* clássicos.

Wazlawick (2004) propõe uma maneira interessante de aplicar esse padrão que consiste em considerar uma classe controladora do sistema no modelo de domínio do problema, relacionando-a a todos os conceitos independentes, correspondentes aos cadastros do sistema, ou seja, os elementos a serem cadastrados para a operação do sistema[22]. O fluxo de controle sempre inicia em uma instância da classe controladora. Essa classe recebe as requisições da interface e, para tratá-las, o controlador invoca métodos das classes do domínio do problema, em uma abordagem chamada de delegação. A delegação consiste em capturar uma operação que trata uma mensagem em um objeto e reenviar essa mensagem para um outro objeto associado ao primeiro que seja capaz de tratar a requisição. Neste caso, a execução é delegada para objetos do domínio do problema, procurando efetuar uma cadeia de delegação sobre as linhas de visibilidade das associações já existentes, até se atingir um objeto capaz de atender à requisição. Novas linhas de visibilidade só devem ser criadas quando isso for estritamente necessário. Deste modo, mantém-se fraco o acoplamento entre as classes, conforme preconiza o padrão de projeto acoplamento fraco (LARMAN, 2004). Assim, partindo-se do controlador do sistema, deve-se identificar qual o objeto a ele relacionado que melhor pode tratar a requisição e delegar essa responsabilidade a ele. Esse objeto, por sua vez, vai colaborar com outros objetos para a realização da funcionalidade.

De maneira geral, um objeto só deve mandar mensagens para outros objetos que estejam a ele associados ou que foram passados como parâmetro no método que está sendo executado. Nessa abordagem, deve-se evitar obter um objeto como retorno de um método (visibilidade local) para mandar mensagens a ele (WAZLAWICK, 2004), conforme apontado pelo padrão "não fale com estranhos" (LARMAN, 2004).

Padrão Camada de Serviço

A motivação principal para esse padrão é o fato de algumas funcionalidades não serem facilmente distribuídas nas classes de domínio do problema, principalmente aquelas que operam sobre vários objetos. Uma possibilidade para resolver tal problema é pulverizar esse comportamento ao longo dos vários objetos do domínio do problema, conforme preconiza o padrão Modelo de Domínio. Contudo, esta pode não ser uma boa solução segundo uma perspectiva de manutenibilidade. Uma alteração em tal funcionalidade poderia afetar diversos objetos e, assim, ser difícil de ser incorporada.

[22] Vale ressaltar que, embora a classe controladora de sistema seja modelada no diagrama de classes do domínio do problema, ela corresponde, de fato, ao pacote de interface com o usuário, tendo em vista que ela é um controlador no sentido adotado pelo padrão MVC.

Uma abordagem alternativa consiste em criar classes controladoras de casos de uso (gerenciadores ou coordenadores de tarefas)[23], responsáveis pela realização de tarefas sobre um determinado conjunto de objetos. Tipicamente, esses gerenciadores agem como aglutinadores, unindo outros objetos para dar forma a um caso de uso. Consequentemente, gerenciadores de tarefa são normalmente encontrados diretamente a partir dos casos de uso.

Os tipos de funcionalidade tipicamente atribuídos a gerenciadores de tarefa incluem comportamento relacionado a transações e sequências de controle específicas de um caso de uso.

O padrão Camada de Serviço (FOWLER, 2003) define uma fronteira da aplicação usando uma camada de serviços que estabelece um conjunto de operações disponíveis e coordena as respostas da aplicação para cada uma das operações. A camada de serviço encapsula a lógica de negócio da aplicação, controlando transações e coordenando respostas na implementação de suas operações. A argumentação em favor desse padrão é que misturar lógica de domínio e lógica de aplicação nas mesmas classes torna essas classes menos reutilizáveis transversalmente em diferentes aplicações, bem como pode dificultar a manutenção da lógica de aplicação, uma vez que a lógica dos casos de uso não é diretamente perceptível em nenhuma classe.

A identificação das operações necessárias na camada de serviço é fortemente apoiada nos casos de uso do sistema. Uma opção é considerar que cada caso de uso vai dar origem a uma classe de serviços, dita classe controladora de caso de uso. Por exemplo, um caso de uso de cadastro, envolvendo funcionalidades de inclusão, alteração, consulta e exclusão, pode ser mapeado em uma classe com operações para tratar essas funcionalidades. Contudo, não há uma prescrição clara, apenas heurísticas. Para uma aplicação relativamente pequena, pode ser suficiente ter uma única classe provendo todas as operações. Para sistemas maiores, compostos de vários subsistemas, pode-se ter uma classe por subsistema (FOWLER, 2003).

A camada de serviço pode ser implementada de duas formas básicas:

- Fachada de Domínio: nessa abordagem, a camada de serviço é implementada como um conjunto fino de fachadas sobre um modelo de domínio (no sentido do padrão Modelo de Domínio). As classes implementando as fachadas não implementam nenhuma lógica de negócio. Esta é implementada pela camada de domínio. As fachadas estabelecem apenas uma fronteira e um conjunto de operações através das quais suas camadas clientes (tipicamente interfaces com o usuário e pontos de integração com outros sistemas) vão interagir com a lógica de negócio.

- Script de Operação: nessa abordagem, a camada de serviço é implementada como um conjunto de classes que implementa diretamente a lógica de aplicação. Contudo, vale frisar que, para implementar a lógica de aplicação, as classes dessa camada delegam diversas responsabilidades para os objetos do domínio do problema.

Não se deve confundir uma abordagem de Camada de Serviços com uma abordagem de Modelo de Domínio Anêmico (*Anemic Domain Model*) (FOWLER, 2003a), na qual os objetos do domínio do problema apresentam comportamento vazio.

[23] Vale lembrar que classes gerenciadoras de casos de uso não são controladores no sentido do padrão MVC.

Nessa abordagem, as classes de análise são divididas em classes de dados (ditos objetos de valor – *Value Objects* – VOs) e classes de lógica (ditos objetos de negócio – *Business Objects* – BOs), que separam o comportamento do estado dos objetos. Os VOs têm apenas o comportamento básico para alterar e manipular seu estado (métodos construtor e destrutor e métodos *get* e *set*). Os BOs ficam com os outros comportamentos, tais como cálculos, validações e regras de negócio. De maneira geral, a abordagem de Modelo de Domínio Anêmico deve ser evitada, sendo, por isso, considerada um anti-padrão. Essa abordagem tem diversos problemas. Primeiro, não há encapsulamento, já que dificilmente um VO vai ser utilizado apenas por um BO. Segundo, a vantagem de se ter um modelo de domínio rico é anulada, já que a proximidade com as abstrações do mundo real é destruída. No mundo real não existe lógica de um lado e dados de outro, mas sim ambos combinados em um mesmo conceito. Outro problema é a manutenção de um sistema construído desta maneira. Os BOs possuem um acoplamento muito alto com os VOs e a mudança em um afeta drasticamente o outro (FRAGMENTAL, 2007).

Uma maneira de se implementar o padrão Camada de Serviço consiste em ter uma ou mais classes controladoras de casos de uso (veja discussão na seção 4.4), as quais encapsulam a lógica da aplicação. Para realizar um caso de uso, a classe controladora de caso de uso invoca métodos da camada de domínio do problema, tal como ocorre no padrão Modelo de Domínio. A diferença entre os dois padrões reside, neste caso, no fato da classe gerenciadora de tarefa centralizar o controle do caso de uso, evitando delegar responsabilidades a classes que não têm como tratá-las. Para tal, aceita-se que a classe gerenciadora de tarefa obtenha um objeto como retorno de um método (visibilidade local) e mande mensagens a ele. Assim, a classe gerenciadora de tarefa pode ter referência a diversos objetos do domínio, tipicamente aqueles envolvidos na realização do caso de uso correspondente.

6.3.2 Componente de Domínio do Problema (CDP)

No projeto orientado a objetos, os modelos conceituais estruturais (diagramas de classes) produzidos na fase de análise fazem parte do Componente de Domínio do Problema (CDP). Como ponto de partida para a elaboração do diagrama de classes do CDP, deve-se utilizar uma cópia do diagrama de classes de análise. A partir dessa cópia, alterações serão feitas para incorporar as decisões de projeto. Vale ressaltar que o trabalho deve ser efetuado em uma cópia, mantendo o modelo conceitual original intacto para efeito de documentação e manutenção do sistema.

Para se poder conduzir o projeto do CDP de maneira satisfatória, algumas informações acerca da plataforma de implementação são essenciais, dentre elas a linguagem de programação e o mecanismo de persistência de objetos a serem adotados. Além disso, informações relativas aos requisitos não funcionais e suas prioridades são igualmente vitais para se tomar decisões importantes relativas ao projeto do CDP.

As alterações básicas a serem incorporadas em um diagrama de classes do CDP são:

- *Adição de informações relativas a tipos de dados de atributos*: Na fase de análise, é comum não especificar tipos de dados para atributos. Na fase de projeto, contudo, essa é uma informação imprescindível. De modo geral, os atributos são mapeados em variáveis de um tipo de dados provido pela linguagem de implementação,

tal como string, inteiro ou booleano. Contudo, muitas vezes, atributos podem dar origem a novas classes (ou tipos de dados enumerados) para atender a requisitos de qualidade, tais como usabilidade, manutenibilidade e reusabilidade.

- *Adição de navegabilidades nas associações*: Na fase de análise, as associações são consideradas navegáveis nos dois sentidos. O mesmo não ocorre na fase de projeto. Pode-se definir que certas associações são navegáveis apenas em um sentido, indicando que apenas um dos objetos terá uma referência para o outro (ou para coleções de objetos, no caso de associações com multiplicidade *). Esta decisão pode ser influenciada pelo mecanismo de persistência de objetos a ser adotado, sobretudo quando esse mecanismo envolve um Sistema Gerenciador de Banco de Dados Relacional.

- *Adição de informações de visibilidade de atributos e associações*: De maneira geral, na fase de análise não se especifica a visibilidade de atributos e associações. Como discutido no item acima, as associações são tipicamente consideradas navegáveis e visíveis nos dois sentidos. Já os atributos são considerados públicos. Porém essa não é uma boa estratégia para a fase de projeto. Ocultar informações é um importante princípio de projeto. Assim, atributos só devem poder ser acessados pela própria classe ou por suas subclasses. Uma classe não deve ter acesso aos atributos de uma classe a ela associada. Como consequência disso, cada classe deve ter operações para consultar (tipicamente nomeadas como *get*) e atribuir / alterar valor (normalmente nomeada como *set*) de cada um de seus atributos e associações navegáveis. Essas operações, contudo, não precisam ser mostradas no diagrama de classes, visto que elas podem ser deduzidas pela própria existência dos atributos e associações (WAZLAWICK, 2004).

- *Adição de métodos às classes*: Muitas vezes, as classes de um diagrama de classes de análise não têm informação acerca das suas operações. Mesmo quando elas têm, essa informação pode ser insuficiente, tendo em vista que é no projeto que se decide efetivamente como abordar a distribuição de responsabilidades para a realização de funcionalidades. Assim, durante o projeto do CDP atenção especial deve ser dada à definição de métodos nas classes. Para apoiar esta etapa, diagramas de sequência podem ser utilizados para modelar a interação entre objetos na realização de funcionalidades do sistema. A escolha de um padrão arquitetônico para o projeto do CDP também tem influência na distribuição de responsabilidades. Vale ressaltar que já se assume que algumas operações, consideradas básicas, existem e, portanto, não precisam ser representadas no diagrama de classes. Essas operações são as operações de criação e destruição de instâncias, além das operações de consulta e atribuição / alteração de valores de atributos e associações, conforme discutido no item anterior. No diagrama de classes devem aparecer apenas os métodos que não podem ser deduzidos (WAZLAWICK, 2004).

- *Eliminação de classes associativas*: Caso o diagrama de classes de análise contenha classes associativas, recomenda-se substituí-las por classes normais, criando novas associações. Isso é importante, pois as linguagens de programação não têm construtores capazes de implementar diretamente esses elementos de modelo.

Além das alterações básicas a que todos os diagramas de classes do CDP estarão sujeitos, outras fontes de alteração incluem:

- *Reutilizar projetos anteriores e classes já programadas*: é importante que na fase de projeto seja levada em conta a possibilidade de se reutilizar classes já projetadas e programadas (desenvolvimento com reúso), bem como a possibilidade de se desenvolver classes para reutilização futura (desenvolvimento para reúso). Tipicamente,

ajustes feitos para incorporar tais classes envolvem alterações na estrutura do modelo, podendo atingir hierarquias de generalização-especialização do modelo, de modo a tratar as classes do domínio do problema como subclasses de classes de biblioteca pré-existentes. Também ao incorporar um padrão de projeto (*design pattern*), muito provavelmente a estrutura do diagrama de classes de projeto sofrerá alterações.

- *Ajustar hierarquias de generalização-especialização*: muitas vezes, as hierarquias de herança da fase de análise não são adequadas para a fase de projeto. Dentre os fatores que podem provocar mudanças na hierarquia de herança destacam-se o mecanismo de herança suportado pela linguagem de programação a ser usada na implementação e a definição de operações.
    - *Ajustar hierarquias de generalização-especialização para adequação ao mecanismo de herança suportado pela linguagem de programação a ser usada na implementação*: se, por exemplo, o modelo de análise envolve herança múltipla e a linguagem de implementação não oferece tal recurso, alterações no modelo são necessárias. Quando se estiver avaliando hierarquias de classes para eliminar relações de herança múltipla, deve-se considerar se uma abordagem de delegação não é mais adequada do que o estabelecimento de uma relação de herança.
    - *Ajustar hierarquias de generalização-especialização para aproveitar oportunidades decorrentes da definição de operações*: as definições de operações nas classes podem também conduzir a alterações na hierarquia de generalização-especialização. De fato, pode ser que durante a fase de análise não sejam exploradas todas as oportunidades de herança. É útil reexaminar o diagrama de projeto procurando observar se determinadas classes têm comportamento parcialmente comum, abrindo-se espaço para a criação de uma superclasse encapsulando as propriedades (atributos e operações) compartilhadas, abstraindo o comportamento comum. Conforme discutido anteriormente, a reutilização pode ser um fator motivador para a criação de novas superclasses. Contudo, deve-se tomar cuidado com a refatoração da hierarquia de classes. Criar uma nova classe para abstrair comportamento comum somente se justifica quando há, de fato, uma relação de subtipo entre as classes existentes e a nova classe criada; ou seja, pode-se dizer que a subclasse é semanticamente um subtipo da superclasse. Não se deve alterar a hierarquia de classes simplesmente para herdar uma parte do comportamento, quando as classes envolvidas não guardam entre si uma relação efetivamente de subtipo, em uma abordagem de herança de implementação (BLAHA; RUMBAUGH, 2006).

- *Ajustar o modelo para melhorar o desempenho*: Visando melhorar o desempenho do sistema, o projetista pode alterar o diagrama de classes do CDP para melhor acomodar os ajustes necessários. Atributos e associações redundantes podem ser adicionados para evitar recomputação, bem como podem ser criadas novas classes para registrar estados intermediários de um processo.

- *Ajustar o modelo para facilitar o projeto de interfaces com o usuário amigáveis*: com o objetivo de incorporar o atributo de qualidade usabilidade, pode ser importante considerar novas classes (ou tipos enumerados de dados) que facilitem a apresentação de listas para seleção do usuário.

- *Ajustar o modelo para incorporar aspectos relacionados à segurança*: táticas como autenticação e autorização requerem novas funcionalidades que, por sua, vez, requerem novas classes do CDP. Em casos como esse, pode ser útil separar as classes relativas a essas funcionalidades em um novo pacote, visando ao reúso.

Além dos ajustes discutidos anteriormente, vários deles relacionados a atributos de qualidade (a saber, reusabilidade, desempenho, usabilidade e segurança), o CDP pode ser alterado, ainda, para comportar outros requisitos não funcionais, tais como testabilidade, confiabilidade etc.

Como dito anteriormente, O CDP é um componente obrigatório, tanto quando se adota o padrão Modelo de Domínio quanto quando se adota o padrão Camada de Serviço. No padrão Modelo de Domínio, o CDP é a própria camada de lógica de negócio, tendo em vista que não há classes gerenciadoras de tarefas (controladoras de casos de uso). No caso do padrão Camada de Serviço, além do CDP, a camada de lógica de negócio tem outro componente, o Componente de Gerência de Tarefas, discutido a seguir.

6.3.3 Componente de Gerência de Tarefas

O Componente de Gerência de Tarefas (CGT) compreende a definição das classes gerenciadoras de tarefas e seu projeto está intimamente relacionado ao modelo de casos de uso.

Em um esboço preliminar, pode-se atribuir um gerenciador de tarefa para cada caso de uso, sendo que os seus fluxos de eventos principais dão origem a operações da classe que representa o caso de uso (classe controladora de caso de uso). Se a abordagem de Script de Operação do padrão Camada de Serviço for adotada, a manutenibilidade pode ser facilitada, uma vez que, detectado um problema em um caso de uso, é fácil identificar a classe que trata do mesmo. Um possível problema, contudo, é o desempenho: para sistemas grandes, com muitos casos de uso, haverá muitas classes de gerência de tarefa e, para realizar uma tarefa complexa, pode ser necessária muita comunicação entre essas classes.

Uma solução diametralmente oposta consiste em definir uma única classe de aplicação para todo o sistema. Neste caso, os cenários de todos os casos de uso dão origem a operações dessa classe. Fica evidente que, exceto para sistemas muito pequenos, essa classe tende a ter muitas operações. Caso a abordagem de Script de Operação do padrão Camada de Serviço seja adotada, essa classe será extremamente complexa e, portanto, essa opção tende a não ser prática. Quando a abordagem de Fachada de Domínio do padrão Camada de Serviço é utilizada, ainda que a classe gerenciadora de tarefas tenha muitas operações, isso pode não ser um problema efetivamente, uma vez que essa classe apenas delega a responsabilidade pela execução das operações para objetos do domínio do problema.

No caso da abordagem de Script de Operação do padrão Camada de Serviço, normalmente, uma solução intermediária entre as duas anteriormente apresentadas conduz a melhores resultados. Nessa abordagem, casos de uso complexos são designados a classes de gerência de tarefas específicas. Casos de uso mais simples e de alguma forma relacionados são tratados por uma mesma classe de gerência de tarefas.

O conjunto de tarefas a serem apoiadas pelo sistema oferece um recurso bastante útil para a definição das janelas, menus e outros componentes de interface com o

usuário necessários para cada uma dessas tarefas. Assim os projetos dos componentes de gerência de tarefa e de apresentação (seção adiante) estão bastante relacionados e devem ser realizados conjuntamente, uma vez que, muitas vezes, são as tarefas que determinam a necessidade de elementos de interface com o usuário para sua execução.

6.4 A Camada de Interface com o Usuário (CIU)

Sistemas, em especial os sistemas de informação, são desenvolvidos para serem utilizados por pessoas. Assim, um aspecto fundamental no projeto de sistemas é a interface com o usuário (IU). O projeto da IU estabelece uma forma de comunicação entre as pessoas e o sistema computacional. A IU define como um usuário comandará o sistema e como o sistema apresentará as informações a ele.

A camada de IU envolve dois tipos de funcionalidades:

- Visão: refere-se aos objetos gráficos usados na interação com o usuário;
- Controle de Interação: diz respeito ao controle da lógica da interface, envolvendo a ativação dos objetos gráficos (p.ex., abrir ou fechar uma janela, habilitar ou desabilitar um item de menu etc.) e o disparo de ações.

Um dos princípios fundamentais para um bom projeto de software é a separação da apresentação (camada de IU) da lógica de negócio. Essa separação é importante por diversas razões, dentre elas (FOWLER, 2003):

- O projeto de IU e o projeto da lógica de negócio tratam de diferentes preocupações. No primeiro, o foco está nos mecanismos de interação e em como dispor uma boa IU. O segundo concentra-se em conceitos e políticas de negócio.
- Usuários podem querer ver as mesmas informações de diferentes maneiras (p.ex., usando diferentes interfaces, tais como interfaces ricas de sistemas *desktop*, navegadores *Web*, interfaces de linha de comando etc.). Neste contexto, separar a IU da lógica de negócio permite o desenvolvimento de múltiplas apresentações.
- Objetos não visuais são geralmente mais fáceis de testar do que objetos visuais. Ao separar objetos da lógica de negócio de objetos de IU, é possível testar os primeiros sem envolver os últimos.

Dada a importância dessa separação, é importante usar algum padrão arquitetônico que trabalhe essa separação, tal como o padrão **Modelo-Visão-Controlador** (**MVC**) (FOWLER, 2003).

6.4.1 O Padrão Modelo-Visão-Controlador (MVC)

O padrão Modelo-Visão-Controlador (MVC) considera três papéis relacionados à interação humano-computador. O *modelo* refere-se aos objetos que representam alguma informação sobre o negócio e corresponde, de fato, a objetos da camada de Lógica de Negócio. A *visão* refere-se à entrada e a exibição de informações na IU. Qualquer requisição é tratada pelo terceiro papel: o controlador. Este pega a entrada do usuário, envia uma requisição para a camada de lógica de negócio, receber sua resposta e solicita que a visão se atualize conforme apropriado. Assim, a IU é uma combinação

de visão e controlador (FOWLER, 2003). Em outras palavras, elementos da visão representam informações de modelo e as exibem ao usuário, que pode enviar, por meio da visão, requisições ao sistema. Essas requisições são tratadas pelo controlador, que as repassa para classes do modelo. Uma vez alterado o estado dos elementos do modelo, o controlador pode, se apropriado, selecionar outros ou alterar elementos de visão a serem exibidos ao usuário. Assim, o controlador situa-se entre o modelo e a visão, isolando-os um do outro.

Neste ponto é importante distinguir o controlador do padrão MVC do controlador de caso de uso do Componente de Gerência de Tarefas. Este último representa uma classe da lógica de negócio, que encapsula a lógica de um caso de uso. Já um controlador do padrão MVC é um controlador de interação, ou seja, ele controla a lógica de interface, abrindo e fechando janelas, habilitando ou desabilitando botões, enviando requisições etc. Para diferenciá-los, neste texto utilizamos o termo controlador de interação para designar os controladores de interface.

O padrão MVC trabalha dois tipos de separação. Primeiro, separa a apresentação (visão) da lógica de negócio (modelo), conforme advogado pelas boas práticas de projeto. Segundo, mantém também separados o controlador e a visão. Essa segunda separação (entre a visão e o controlador) é menos importante que a primeira (entre a visão e a lógica de negócio). A maioria dos sistemas tem um único controlador por visão e, por isso, a separação entre a visão e o controlador muitas vezes não é feita. Em sistemas de interfaces ricas *desktop* ela é muitas vezes desprezada. Contudo, em interfaces Web, essa separação é comum, já que a parte de visão *front end* é naturalmente separada do controlador. De fato, a maioria dos padrões de projeto de interfaces Web é baseada nesse princípio (FOWLER, 2003). A Figura 5.2 mostra um diagrama de pacotes ilustrando o padrão MVC.
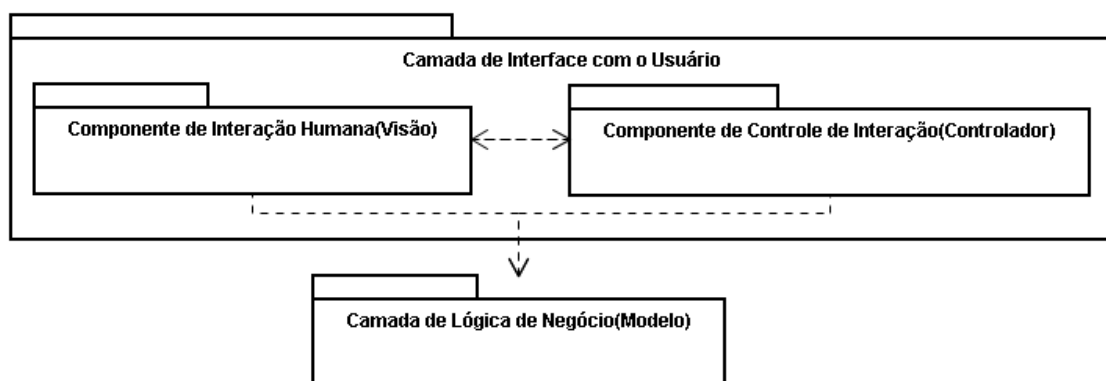
Figura 6.4 – O Padrão MVC.

A separação entre visão e controlador dá origem a dois tipos de classes que podem ser organizados em dois pacotes na camada de interface com o usuário: o Componente de Interação Humana (CIH), que é responsável pelas interfaces com o usuário propriamente ditas (janelas, painéis, botões, menus etc.) e representa a visão no modelo MVC; e o Componente de Controle de Interação (CCI), que é responsável por controlar a interação, recebendo requisições da interface, disparando operações da lógica de negócio e atualizando a visão com base no retorno dessas operações. O CCI é, portanto, o controlador do modelo MVC.

É importante frisar que, mesmo quando se opta por não fazer a separação física em pacotes de visão e controlador, é útil ter classes distintas para desempenhar esses

papéis. As classes controladoras de interação devem ser marcadas com o estereótipo <<control>> para diferenciá-las das classes de visão, que devem ser marcadas com o estereótipo <<boundary>>.

No que se refere à interação entre as camadas de IU e Lógica de Negócio (modelo), ela se dá de maneiras distintas em função do padrão arquitetônico adotado nesta última. Quando o padrão Modelo de Domínio é adotado, os controladores de interação enviam as requisições diretamente para os objetos do domínio do problema (CDP), uma vez que neste caso não existem objetos gerenciadores de tarefa (CGT). Quando o padrão Camada de Serviço é adotado, as requisições dos controladores de interação são enviadas para os objetos gerenciadores de tarefas (CGT).

6.4.2 Componente de Interação Humana (CIH)

A porção do sistema que lida com a visão da interface com o usuário deve ser mantida tão independente e separada do resto da arquitetura do software quanto possível. Aspectos de interface com o usuário provavelmente serão alvo de alterações ao longo da vida do sistema e essas alterações devem ter um impacto mínimo nas demais partes do sistema.

O Componente de Interação Humana (CIH) trata do projeto da interação humano-computador, definindo formato de janelas, formulários, relatórios, entre outros. Durante o projeto do CIH, é muito útil construir protótipos, de modo a apoiar a seleção e o desenvolvimento dos mecanismos de interação a serem usados.

Como discutido anteriormente, o ponto de partida para o projeto da CIH é o modelo de casos de uso e as descrições de atores e casos de uso. Com base nos casos de uso, deve-se projetar uma hierarquia de comandos, definindo barras de menus, menus *pull-down*, ícones etc., que levem à execução dos casos de uso quando acionados pelo usuário. A hierarquia de comandos deve respeitar convenções e estilos existentes com os quais o usuário já esteja familiarizado. Note que a hierarquia de comandos é, de fato, um meio de apresentar ao usuário as várias funcionalidades disponíveis no sistema. Assim, a hierarquia de comandos deve permitir o acesso aos casos de uso do sistema.

Uma vez definida a hierarquia de comandos, as interações detalhadas entre o usuário e o sistema devem ser projetadas. Neste momento, é útil observar atentamente táticas de usabilidade, discutidas logo adiante.

Normalmente, não é necessário projetar as classes básicas de interfaces gráficas com o usuário. Existem vários ambientes de desenvolvimento de interfaces, oferecendo classes reutilizáveis (janelas, ícones, botões etc.) e, portanto, basta especializar as classes e instanciar os objetos que possuem as características apropriadas para o problema em questão. Hierarquias de classes de visão podem ser desenvolvidas visando à uniformidade da apresentação e ao reúso.

Táticas de Usabilidade

Diversas táticas relativas à usabilidade podem ser aplicadas durante o projeto de IU. Algumas dessas táticas gerais incluem:

- Facilidade de Ajuda: Ajuda é fundamental para os usuários, sobretudo aqueles novatos ou conhecedores, mas esporádicos. Para projetar adequadamente uma facilidade de ajuda é necessário definir, dentre outros:
    - quando a ajuda estará disponível e para que funções do sistema;
    - como ativar (botão, tecla de função, menu);
    - como representar (janela separada, local fixo da tela);
    - como retornar à interação normal (botão, tecla de função);
    - como estruturar a informação (estrutura plana, hierárquica, hipertexto).
- Mensagens de Erro e Avisos: Mais até do que a ajuda, as mensagens de erro e avisos são fundamentais para uma boa interação humano-computador. Ao definir mensagens de erro e avisos considere as seguintes diretrizes:
    - Descreva o problema com um vocabulário passível de entendimento pelo usuário.
    - Sempre que possível, proveja assistência para recuperar o erro.
    - Quando for o caso, indique as consequências negativas do erro.
    - Para facilitar a percepção da mensagem por parte do usuário, pode ser útil que a mesma seja acompanhada de uma dica visual ou sonora.
    - Não censure o usuário.
- Tipos de Comandos: Diferentes grupos de usuários têm diferentes necessidades de interação. Em muitas situações é útil prover ao usuário mais de uma forma de interação. Nestes casos, é necessário definir e avaliar:
    - se toda opção de menu terá um comando correspondente;
    - a forma do comando, tais como controle de sequência (p.ex., ^Q), teclas de função (p.ex., F1) e comandos digitados;
    - quão difícil é aprender e lembrar o comando;
    - possibilidade de customização de comandos (macros);
    - padrões para todo sistema e conformidade com outros padrões, tal como o definido pelo sistema operacional ou por produtos de software tipicamente utilizados pelos usuários.
- Tempo de Resposta: É importante mostrar o progresso do processamento para os usuários, principalmente para eventos com tempo de resposta longo ou com grande variação de tempos de resposta.

Levando-se em conta princípios gerais de projeto de IU, algumas orientações adicionais devem ser consideradas, dentre elas:

- Seja consistente. Use formatos consistentes para seleção de menus, entrada de comandos, apresentação de dados etc.
- Ofereça retorno significativo ao usuário.

- Peça confirmação para ações destrutivas, tais como ações para apagar ou sobrepor informações, terminar a seção corrente do aplicativo.
- Permita reversão fácil da maioria das ações (função *Desfazer*).
- Reduza a quantidade de informação que precisa ser memorizada entre ações.
- Busque eficiência no diálogo (movimentação, teclas a serem apertadas).
- Trate possíveis erros do usuário. O sistema deve se proteger de erros, casuais ou não, provocados pelo usuário.
- Classifique atividades por função e organize geograficamente a tela de acordo. Menus do tipo *pull-down* são uma boa opção.
- Proveja facilidades de ajuda sensíveis ao contexto.
- Use verbos de ação simples ou frases curtas para nomear funções e comandos.

No que se refere à apresentação de informações, considere as seguintes diretrizes:

- Mostre apenas informações relevantes ao contexto corrente.
- Use formatos de apresentação que permitam assimilação rápida da informação, tais como gráficos e figuras.
- Use rótulos consistentes, abreviaturas padrão e cores previsíveis.
- Produza mensagens de erro significativas.
- Projete adequadamente o layout de informações textuais. Leve em consideração o bom uso de letras maiúsculas e minúsculas, identação, agrupamento de informações etc.
- Separe diferentes tipos de informação. Painéis ou mesmo janelas podem ser usadas para este fim.
- Use formas de representação análogas às do mundo real para facilitar a assimilação da informação. Para tal considere o uso de figuras, cores etc.

No que se refere à entrada de dados, considere as seguintes diretrizes:

- Minimize o número de ações de entrada requeridas e possíveis erros. Para tal considere a seleção de dados a partir de um conjunto pré-definido de valores de entrada, o uso de valores *default* e macros etc.
- Mantenha consistência entre apresentação e entrada de dados; ou seja, mantenha as mesmas características visuais, dentre elas tamanho do texto, cor e localização.
- Permita ao usuário customizar a entrada para seu uso, quando possível, dando-lhe liberdade para definir comandos customizados, dispensar algumas mensagens de aviso e verificações de ações, dentre outros.
- Flexibilize a interação, permitindo afiná-la ao modo de entrada preferido do usuário (comandos, botões, *plug-and-play*, digitação etc.).

- Desative comandos inapropriados para o contexto das ações correntes.
- Proveja ajuda significativa para assistir as ações de entrada de dados.
- Nunca requeira que o usuário entre com uma informação que possa ser adquirida automaticamente pelo sistema ou computada por ele.

6.4.3 – Componente de Controle de Interação (CCI)

O Componente de Controle de Interação (CCI) trata das classes responsáveis por controlar a interação (ativação / desativação dos objetos do CIH) e enviar requisições para os objetos da Lógica de Negócio.

Em sistemas rodando em plataforma *desktop*, deve haver pelo menos uma classe controladora, dita classe controladora de sistema, representando o sistema como um todo. Os objetos dessa classe representam as várias sessões (execuções) do sistema. Neste caso, é necessário levar em conta, ainda, quantos executáveis devem ser gerados para o sistema. Se mais do que um for necessário, cada executável terá de dar origem a uma classe controladora. Esta, contudo, é apenas uma abordagem possível. Analogamente ao projeto do CGT (veja seção 4.4), é possível definir um número arbitrário de controladores de interação. Uma opção em linha com o projeto da CGT é definir um controlador de interação para cada caso de uso. Contudo, uma vez que as classes controladoras de interação tendem a ser muito simples, essa abordagem pode ser exagerada. Assim, ainda que haja uma analogia entre o projeto do CCI e o projeto do CGT, as motivações são bastante diferentes e a escolha dos controladores de interação tende a ser diferente da escolha dos gerenciadores de tarefas.

6.5 A Camada de Gerência de Dados (CGD)

A maioria dos sistemas requer alguma forma de armazenamento de dados. Para tal, há várias alternativas, dentre elas a persistência em arquivos e bancos de dados. Em especial os sistemas de informação envolvem grandes quantidades de dados e fazem uso de sistemas gerenciadores de bancos de dados (SGBDs). Há diversos tipos de SGBDs, dentre eles os relacionais e os orientados a objetos, sendo os primeiros os mais utilizados atualmente no desenvolvimento de sistemas de informação.

Quando SGBDs Relacionais são utilizados, é necessário um mapeamento entre as estruturas de dados dos modelos orientado a objetos e relacional, de modo que objetos possam ser armazenados em tabelas. Dentre as principais diferenças entre esses modelos, destacam-se as diferentes formas como objetos e tabelas tratam ligações e na ausência do mecanismo de herança no modelo relacional. Essas diferenças levam à necessidade de reversões das estruturas de dados entre objetos e tabelas, tratadas como mapeamento objeto-relacional.

Além das diferenças estruturais, outros aspectos têm de ser tratados durante o projeto da persistência, dentre eles o modo como a camada de lógica de negócio se comunica com o banco de dados, o problema comportamental que diz respeito a como obter vários objetos do banco e como salvá-los, e o tratamento de conexões com o banco de dados e transações (FOWLER, 2003).

É importante enfatizar que muitos desses problemas são tratados por *frameworks* de persistência de objetos em bancos de dados relacionais (ou *frameworks* de

mapeamento objeto-relacional), tal como o Hibernate[24]. Os desenvolvedores desses frameworks têm despendido muitos esforços trabalhando nesses problemas e tais ferramentas são bem mais sofisticadas do que a maioria das soluções específicas que podem ser construídas à mão. Contudo, mesmo quando um framework de mapeamento objeto-relacional (O/R) é utilizado, é importante estar ciente dos padrões usados. Boas ferramentas de mapeamento O/R dão várias opções de mapeamento para um banco de dados e esses padrões vão ajudá-lo a entender quando selecionar as diferentes opções (FOWLER, 2003).

A seguir asão apresentados alguns padrões para o projeto do **Componente de Gerência de Dados.**

6.5.1 – Padrões Arquitetônicos para a Camada de Gerência de Dados

A Camada de Persistência (ou Camada de Gerência de Dados ou, ainda, **Componente de Gerência de Dados - CGD**) provê a infraestrutura básica para o armazenamento e a recuperação de objetos no sistema. Sua finalidade é isolar os impactos da tecnologia de gerenciamento de dados sobre a arquitetura do software (COAD; YOURDON, 1993).

A despeito da opção de persistência adotada (SGBD relacional, SGBD orientado a objetos, arquivos), há uma importante questão a ser considerada no projeto do CGD: Que classes devem suportar a persistência dos objetos?

Uma abordagem consiste em isolar completamente a lógica de negócio e o banco de dados, criando uma camada responsável pelo mapeamento entre objetos do domínio e tabelas do banco de dados. Os padrões Mapeador de Dados (*Data Mapper*) (FOWLER, 2003) e Objeto de Acesso a Dados (*Data Access Object* - DAO) (BAUER; KING, 2007) adotam esta filosofia, de modo que apenas uma parte da arquitetura de software fica ciente da tecnologia de persistência adotada. Essa parte, o Componente de Gerência de Dados (CGD), serve como uma camada intermediária separando objetos do domínio de objetos de gerência de dados. Via conexões de mensagem, o CGD lê e escreve dados, estabelecendo uma comunicação entre a base de dados e os objetos do sistema. Qualquer código SQL[25] está confinado nessas classes, de modo que não código desse tipo em outras classes da arquitetura do software.

O Padrão Data Mapper

O padrão Mapeador de Dados (FOWLER, 2003) prescreve uma camada de objetos mapeadores que transferem dados entre objetos em memória e o banco de dados, mantendo-os independentes um do outro e dos mapeadores em si. Os objetos em memória não têm qualquer conhecimento acerca do esquema do banco de dados e não precisam de nenhuma interface para código SQL. De fato, eles não precisam saber sequer que há um banco de dados. O banco de dados, por sua vez, desconhece completamente os objetos que o utilizam.

[24] http://www.hibernate.org/
[25] SQL é a abreviatura de *Structured Query Language* (Linguagem Estruturada de Consulta), a linguagem de consulta dos bancos de dados relacionais.

Em sua versão mais simples, para cada classe a ser persistida em uma tabela, há uma correspondente classe mapeadora. Seja o exemplo da Figura 6.6. Nesse exemplo, a classe mapeadora *PersonMapper* intermedeia a classe *Person* da lógica de negócio e o acesso a seus dados no banco de dados, na correspondente tabela (FOWLER, 2003).
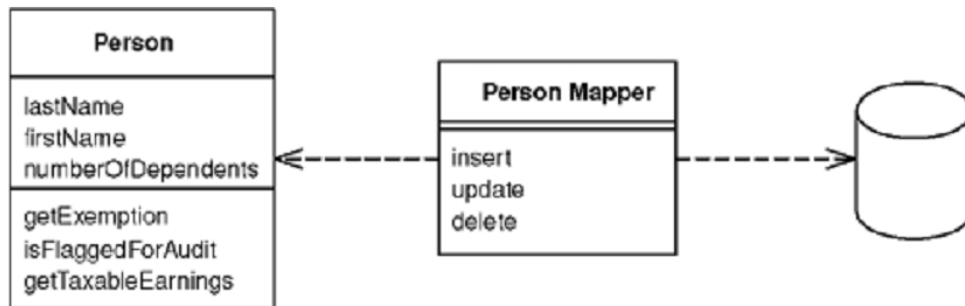
Figura 6.5 – Padrão *Data Mapper* (FOWLER, 2003).

O Padrão DAO

O padrão DAO define uma interface de operações de persistência, incluindo métodos para criar, recuperar, alterar, excluir e diversos tipos de consulta, relativa a uma particular entidade persistente, agrupando o código relacionado à persistência daquela entidade (BAUER; KING, 2007). A estrutura básica do padrão, como proposto em (BAUER; KING, 2007), é apresentada na Figura 6.7.
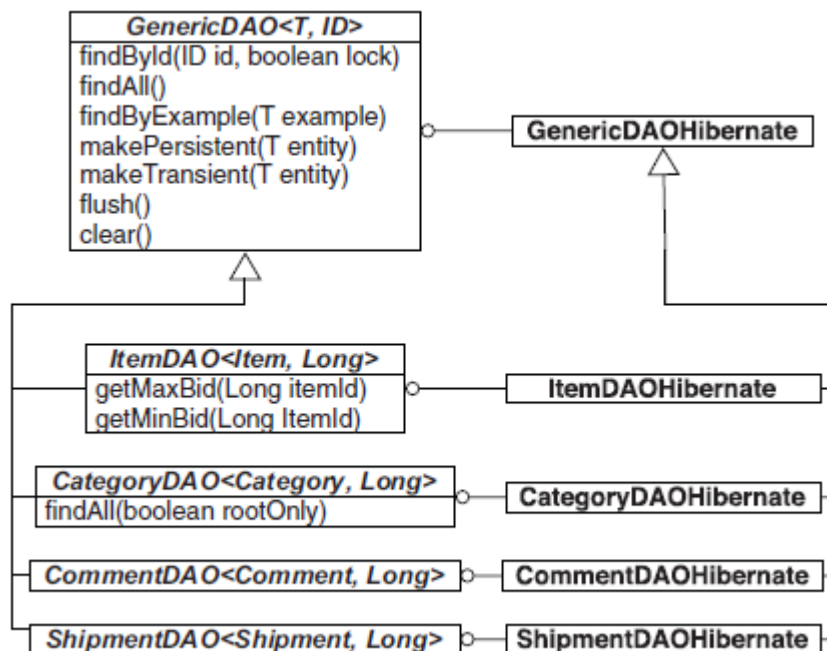
Figura 6.6 – Padrão DAO (BAUER; KING, 2007).

Seguindo esse padrão, a camada de persistência é implementada por duas hierarquias paralelas: interfaces à esquerda e implementações à direita. As operações básicas de armazenamento e recuperação de objetos são agrupadas em uma interface genérica (*GenericDAO*) e uma superclasse genérica (no exemplo da Figura 6.6,

GenericDAOHibernate). Esta última implementa as operações com uma particular solução de persistência (no caso, Hibernate). A interface genérica é estendida por interfaces para entidades específicas que requerem operações adicionais de acesso a dados. O mesmo ocorre com a hierarquia de classes de implementação. Uma característica marcante desta solução é que é possível ter várias implementações de uma mesma interface DAO (BAUER; KING, 2007).

Leitura Complementar

No Capítulo 7 de (WAZLAWICK, 2004) – Projeto da Camada de Domínio, aspectos do projeto da Lógica de Negócio são discutidos, incluindo modelos de domínio ricos e padrões de projeto associados, possíveis modificações a serem feitas nos diagramas de classes de projeto.

Em (FOWLER, 2003), o Capítulo 2 – *Organizing Domain Logic* – discute a organização da camada de lógica de negócio. No Capítulo 9 – *Domain Logic Patterns* são apresentados padrões para a camada lógica de negócio.

Em (FOWLER, 2003), o Capítulo 4 – *Web Presentation* – discute diversos aspectos do projeto da interface com o usuário de aplicações Web. Os padrões discutidos nesse capítulo são posteriormente apresentados em detalhes no Capítulo 14 – *Web Presentation Patterns*.

Em (PRESSMAN, 2006), o Capítulo 12 – Projeto de Interface com o Usuário – discute princípios gerais do projeto da IU (seção 12.1) e o processo de análise, projeto e avaliação de IU (seções 12.2 a 12.5).

O Capítulo 9 de (WAZLAWICK, 2004) – Projeto da Camada de Interface, trata do projeto da camada de Interface com o Usuário focalizando aspectos da navegação do sistema e controle de segurança.

Em (FOWLER, 2003), o Capítulo 3 – *Mapping to Relational Databases* – discute diversos aspectos do projeto da persistência de objetos em bancos de dados relacionais.

O Capítulo 10 de (WAZLAWICK, 2004) – Projeto da Camada de Persistência, trata do projeto da camada de persistência, incluindo equivalências entre modelos de classes e modelos relacionais e aspectos relativos a como e quando os objetos serão salvos e carregados do banco de dados.

Bauer e King (2007) discutem vários aspectos do projeto da camada de persistência, indo desde mapeamento O/R até detalhes de implementação usando o *framework* Hibernate.

Referências do Capítulo

BAUER, C., KING, G., *Java Persistence with Hibernate*, Manning, 2007.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M., *Pattern-Oriented Software Architecture: A System of Patterns*, Volume 1, Wiley, 1996.

COAD, P., YOURDON, E., *Projeto Baseado em Objetos*, Editora Campus, 1993.

FOWLER, M., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.

PRESSMAN, R.S., Engenharia de Software, McGraw-Hill, 6ª edição, 2006.

SOUZA, V.E.S., FrameWeb: um Método baseado em *Frameworks* para o Projeto de Sistemas de Informação *Web*, Dissertação de Mestrado, Programa de Pós-Graduação em Informática, UFES, Universidade Federal do Espírito Santo, 2005.

WAZLAWICK, R.S., *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, Elsevier, 2004.

Capítulo 7 – Implementação e Testes

Uma vez projetado o sistema, é necessário escrever os programas que implementem esse projeto e testá-los.

7.1 - Implementação

Ainda que um projeto bem elaborado facilite sobremaneira a implementação, essa tarefa não é necessariamente fácil. Muitas vezes, os projetistas não conhecem em detalhes a plataforma de implementação e, portanto, não são capazes de (ou não desejam) chegar a um projeto algorítmico passível de implementação direta. Além disso, questões relacionadas à legibilidade, alterabilidade e reutilização têm de ser levadas em conta.

Deve-se considerar, ainda, que programadores, geralmente, trabalham em equipe, necessitando integrar, testar e alterar código produzido por outros. Assim, é muito importante que haja padrões organizacionais para a fase de implementação. Esses padrões devem ser seguidos por todos os programadores e devem estabelecer, dentre outros, padrões de nomes de variáveis, formato de cabeçalhos de programas e formato de comentários, recuos e espaçamento, de modo que o código e a documentação a ele associada sejam claros para quaisquer membros da organização.

Padrões para cabeçalho, por exemplo, podem informar o que o código (programa, módulo ou componente) faz, quem o escreveu, como ele se encaixa no projeto geral do sistema, quando foi escrito e revisado, apoios para teste, entrada e saída esperadas etc. Essas informações são de grande valia para a integração, testes, manutenção e reutilização (PFLEEGER, 2004).

Além dos comentários feitos no cabeçalho dos programas, comentários adicionais ao longo do código são também importantes, ajudando a compreender como o componente é implementado.

Por fim, o uso de nomes significativos para variáveis, indicando sua utilização e significado, é imprescindível, bem como o uso adequado de recuo e espaçamento entre linhas de código, que ajudam a visualizar a estrutura de controle do programa (PFLEEGER, 2004).

Além da documentação interna, escrita no próprio código, é importante que o código de um sistema possua também uma documentação externa, incluindo uma visão geral dos componentes do sistema, grupos de componentes e da inter-relação entre eles (PFLEEGER, 2004).

Ainda que padrões sejam muito importantes, deve-se ressaltar que a correspondência entre os componentes do projeto e o código é fundamental, caracterizando-se como a mais importante questão a ser tratada. O projeto é o guia para a implementação, ainda que o programador deva ter certa flexibilidade para implementá-lo como código (PFLEEGER, 2004).

Como resultado de uma implementação bem-sucedida, as unidades de software devem ser codificadas e critérios de verificação das mesmas devem ser definidos.

7.2 - Testes

Uma vez implementado o código de uma aplicação, o mesmo deve ser testado para descobrir tantos defeitos quanto possível, antes da entrega do produto de software ao seu cliente.

Conforme discutido no Capítulo 4 (Parte I do material), o teste é uma atividade de verificação e validação do software e consiste na análise dinâmica do mesmo, isto é, na execução do produto de software com o objetivo de verificar a presença de defeitos no produto e aumentar a confiança de que o mesmo está correto (MALDONADO e FABBRI, 2001).

Vale ressaltar que, mesmo se um teste não detectar defeitos, isso não quer dizer necessariamente que o produto é um produto de boa qualidade. Muitas vezes, a atividade de teste empregada pode ter sido conduzida sem planejamento, sem critérios e sem uma sistemática bem definida, sendo, portanto, os testes de baixa qualidade (MALDONADO e FABBRI, 2001).

Assim, o objetivo é projetar testes que potencialmente descubram diferentes classes de erros e fazê-lo com uma quantidade mínima de esforço (PRESSMAN, 2006). Ainda que os testes não possam demonstrar a ausência de defeitos, como benefício secundário, podem indicar que as funções do software parecem estar funcionando de acordo com o especificado.

A idéia básica dos testes é que os defeitos podem se manifestar por meio de falhas observadas durante a execução do software. Essas falhas podem ser resultado de uma especificação errada ou falta de requisito, de um requisito impossível de implementar considerando o hardware e o software estabelecidos, o projeto pode conter defeitos ou o código pode estar errado. Assim, uma falha é o resultado de um ou mais defeitos (PFLEEGER, 2004).

São importantes princípios de testes a serem observados (PRESSMAN, 2006; PFLEEGER, 2004):

- Teste completo não é possível, ou seja, mesmo para sistemas de tamanho moderado, pode ser impossível executar todas as combinações de caminhos durante o teste.
- Teste envolve vários estágios. Geralmente, primeiro, cada módulo é testado isoladamente dos demais módulos do sistema (teste de unidade). À medida que os testes progridem, o foco se desloca para a integração dos módulos (teste de integração), até se chegar ao sistema como um todo (teste de sistema).
- Teste deve ser conduzido por terceiros. Os testes conduzidos por outras pessoas que não aquelas que produziram o código têm maior probabilidade de encontrar defeitos. O desenvolvedor que produziu o código pode estar muito envolvido com ele para poder detectar defeitos mais sutis.
- Testes devem ser planejados bem antes de serem realizados. Um plano de testes deve ser utilizado para guiar todas as atividades de teste e deve incluir objetivos do teste, abordando cada tipo (unidade, integração e sistema), como serão executados e quais critérios a serem utilizados para determinar

quando o teste está completo. Uma vez que os testes estão relacionados aos requisitos dos clientes e usuários, o planejamento dos testes pode começar tão logo a especificação de requisitos tenha sido elaborada. À medida que o processo de desenvolvimento avança (análise, projeto e implementação), novos testes vão sendo planejados e incorporados ao plano de testes.

O processo de teste envolve quatro atividades principais (PFLEEGER, 2004, MALDONADO e FABBRI, 2001):

- Planejamento de Testes: trata da definição das atividades de teste, das estimativas dos recursos necessários para realizá-las, dos objetivos, estratégias e técnicas de teste a serem adotadas e dos critérios para determinar quando uma atividade de teste está completa.
- Projeto de Casos de Testes: é a atividade chave para um teste bem-sucedido, ou seja, para se descobrir a maior quantidade de defeitos com o menor esforço possível. Os casos de teste devem ser cuidadosamente projetados e avaliados para tentar se obter um conjunto de casos de teste que seja representativo e envolva as várias possibilidades de exercício das funções do software (cobertura dos testes). Existe uma grande quantidade de técnicas de teste para apoiar os testadores a projetar casos de teste, oferecendo uma abordagem sistemática para o teste de software.
- Execução dos testes: consiste na execução dos casos de teste e registro de seus resultados.
- Avaliação dos resultados: detectadas falhas, os defeitos deverão ser procurados. Não detectadas falhas, deve-se fazer uma avaliação final da qualidade dos casos de teste e definir pelo encerramento ou não de uma atividade de teste.

7.2.1 – Técnicas de Teste

Para testar um módulo, é necessário definir um caso de teste, executar o módulo com os dados de entrada definidos por esse caso de teste e analisar a saída. Um teste é um conjunto limitado de casos de teste, definido a partir do objetivo do teste (PFLEEGER, 2004).

Diversas técnicas de teste têm sido propostas visando apoiar o projeto de casos de teste. Essas técnicas podem ser classificadas, segundo a origem das informações utilizadas para estabelecer os objetivos de teste, em, dentre outras categorias, técnicas funcional, estrutural ou baseadas em máquinas de estado (MALDONADO e FABBRI, 2001).

Os testes funcionais ou caixa-preta utilizam as especificações (de requisitos, análise e projeto) para definir os objetivos do teste e, portanto, para guiar o projeto de casos de teste. O conhecimento sobre uma determinada implementação não é usado (MALDONADO e FABBRI, 2001). Assim, os testes são conduzidos na interface do software. Os testes caixa-preta são empregados para demonstrar que as funções do software estão operacionais, que a entrada é adequadamente aceita e a saída é corretamente produzida e que a integridade da informação externa (uma base de dados, por exemplo) é mantida (PRESSMAN, 2006).

Os testes estruturais ou caixa-branca estabelecem os objetivos do teste com base em uma determinada implementação, verificando detalhes do código. Caminhos lógicos internos são testados, definindo casos de testes que exercitem conjuntos específicos de condições ou laços (PRESSMAN, 2006).

Os testes baseados em máquinas de estado são projetados utilizando o conhecimento subjacente à estrutura de uma máquina de estados para determinar os objetivos do teste (MALDONADO e FABBRI, 2001).

É importante ressaltar que técnicas de teste devem ser utilizadas de forma complementar, já que elas têm propósitos distintos e detectam categorias de erros distintas (MALDONADO e FABBRI, 2001). À primeira vista, pode parecer que realizando testes caixa branca rigorosos poderíamos chegar a programas corretos. Contudo, conforme anteriormente mencionado, isso não é prático, uma vez que todas as combinações possíveis de caminhos e valores de variáveis teriam de ser exercitadas, o que é impossível. Isso não quer dizer, entretanto, que os testes caixa-branca não são úteis. Testes caixa-branca podem ser usados, por exemplo, para garantir que todos os caminhos independentes[26] de um módulo tenham sido exercitados pelo menos uma vez (PRESSMAN, 2006).

Há diversas técnicas de testes caixa-branca, cada uma delas procurando apoiar o projeto de casos de teste focando em algum ou vários aspectos da implementação. Dentre elas, podem ser citadas (PRESSMAN, 2006):

- **Testes de estrutura de controle**: como o próprio nome diz, enfocam as estruturas de controle de um módulo, tais como comandos, condições e laços. Teste de condição é um tipo de teste de estrutura de controle que exercita as condições lógicas contidas em um módulo. Um teste de fluxo de dados, por sua vez, seleciona caminhos de teste tomando por base a localização das definições e dos usos das variáveis nos módulos. Testes de ciclo ou laço focalizam exclusivamente os laços (*loops*).
- **Teste de caminho básico**: define uma medida de complexidade lógica de um módulo e usa essa medida como guia para definir um conjunto básico de caminhos de execução.

Assim como há diversas técnicas de teste caixa-branca, o mesmo acontece em relação ao teste caixa-preta. Dentre as diversas técnicas de teste caixa-preta, podem ser citadas (PRESSMAN, 2006):

- **Particionamento de equivalência**: divide o domínio de entrada de um módulo em classes de equivalência, a partir das quais casos de teste são derivados. A meta é minimizar o número de casos de teste, ficando apenas com um caso de teste para cada classe, uma vez que, a princípio, todos os elementos de uma mesma classe devem se comportar de maneira equivalente.
- **Análise de valor limite**: a prática mostra que um grande número de erros tende a ocorrer nas fronteiras do domínio de entrada de um módulo. Tendo isso em mente, a análise de valor limite leva à seleção de casos de teste que exercitem os valores limítrofes.

[26] Um caminho independente é qualquer caminho ao longo de um módulo que introduz pelo menos um novo comando de processamento ou condição (PRESSMAN, 2006).

7.2.2 – Estratégias de Teste

O projeto efetivo de casos de teste é importante, mas não suficiente para o sucesso da atividade de testes. A estratégia, isto é, a série planejada de realização dos testes, é também crucial (PRESSMAN, 2006). Basicamente, há três grandes fases de teste (MALDONADO e FABBRI, 2001):

- **Teste de Unidade**: tem por objetivo testar a menor unidade do projeto (um componente de software que não pode ser subdividido), procurando identificar erros de lógica e de implementação em cada módulo separadamente. No paradigma estruturado, a menor unidade refere-se a um procedimento ou função.
- **Teste de Integração**: visa a descobrir erros associados às interfaces entre os módulos quando esses são integrados para formar estrutura do produto de software.
- **Teste de Sistema**: tem por objetivo identificar erros de funções (requisitos funcionais) e características de desempenho (requisito não funcional) que não estejam de acordo com as especificações.

Tomando por base essas fases, a atividade de teste pode ser estruturada de modo que, em cada fase, diferentes tipos de erros e aspectos do software sejam considerados (MALDONADO e FABBRI, 2001). Tipicamente, os primeiros testes focalizam componentes individuais e aplicam testes caixa-branca e caixa-preta para descobrir erros. Após os componentes individuais terem sido testados, eles precisam ser integrados, até se obter o sistema por inteiro. Na integração, o foco é o projeto e a arquitetura do sistema. Finalmente, uma série de testes de alto nível é executada quando o sistema estiver operacional, visando a descobrir erros nos requisitos (PRESSMAN, 2006, PFLEEGER, 2004).

No teste de unidade, faz-se necessário construir pequenos componentes para permitir testar os módulos individualmente, os ditos *drivers* e *stubs*. Um *driver* é um programa responsável pela ativação e coordenação do teste de uma unidade. Ele é responsável por receber os dados de teste fornecidos pelo testador, passar esses dados para a unidade sendo testada, obter os resultados produzidos por essa unidade e apresentá-los ao testador. Um *stub* é uma unidade que substitui, na hora do teste, uma outra unidade chamada pela unidade que está sendo testada. Em geral, um *stub* simula o comportamento da unidade chamada com o mínimo de computação ou manipulação de dados (MALDONADO e FABBRI, 2001).

A abordagem de integração de módulos pode ter impacto na quantidade de *drivers* e *stubs* a ser construída. Sejam as seguintes abordagens:

- **Integração ascendente ou *bottom-up***: Nessa abordagem, primeiramente, cada módulo no nível inferior da hierarquia do sistema é testado individualmente. A seguir, são testados os módulos que chamam esses módulos previamente testados. Esse procedimento é repetido até que todos os módulos tenham sido testados (PFLEEGER, 2004). Neste caso, apenas *drivers* são necessários. Seja o exemplo da figura 7.1. Usando a abordagem de integração ascendente, os módulos seriam testados da seguinte forma. Inicialmente, seriam testados os módulos do nível inferior (E, F e G). Para

cada um desses testes, um *driver* teria de ser construído. Concluídos esses testes, passaríamos ao nível imediatamente acima, testando seus módulos (B, C e D) combinados com os módulos por eles chamados. Neste caso, testamos juntos B, E e F bem como C e G. Novamente, três *drivers* seriam necessários. Por fim, testaríamos todos os módulos juntos.
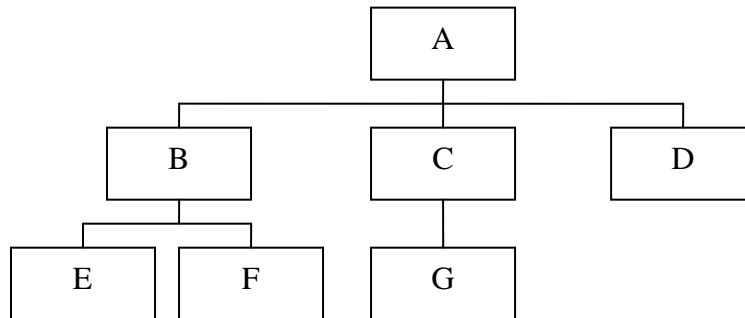
Figura 7.1 – Exemplo de uma hierarquia de módulos.

- **Integração descendente ou *top-down***: A abordagem, neste caso, é precisamente o contrário da anterior. Inicialmente, o nível superior (geralmente um módulo de controle) é testado sozinho. Em seguida, todos os módulos chamados por pelo módulo testado são combinados e testados como uma grande unidade. Essa abordagem é repetida até que todos os módulos tenham sido incorporados (PFLEEGER, 2004). Neste caso, apenas *stubs* são necessários. Tomando o exemplo da figura 7.1, o teste iniciaria pelo módulo A e três *stubs* (para B, C e D) seriam necessários. Na seqüência seriam testados juntos A, B, C e D, sendo necessários *stubs* para E, F e G. Por fim, o sistema inteiro seria testado.

Muitas outras abordagens, algumas usando as apresentadas anteriormente, podem ser adotadas, tal como a integração sanduíche (PFLEEGER, 2004), que considera uma camada alvo no meio da hierarquia e utiliza as abordagens ascendente e descendente, respectivamente para as camadas localizadas abaixo e acima da camada alvo. Outra possibilidade é testar individualmente cada módulo e só depois de testados individualmente integrá-los (teste *big-band*). Neste caso, tanto *drivers* quanto *stubs* têm de ser construídos para cada módulo, o que leva a muito mais codificação e problemas em potencial (PFLEEGER, 2004).

Uma vez integrados todos os módulos do sistema, parte-se para os testes de sistema, quando se busca observar se o software funciona conforme esperado pelo cliente. Por isso mesmo, muitas vezes, os testes de sistema são chamados de testes de validação. Os testes de sistema incluem diversos tipos de teste, realizados na seguinte ordem (PFLEEGER, 2004):

- Teste funcional: verifica se o sistema integrado realiza as funções especificadas nos requisitos;
- Teste de desempenho: verifica se o sistema integrado atende os requisitos não funcionais do sistema (eficiência, segurança, confiabilidade etc);
- Teste de aceitação: os testes funcional e de desempenho são ainda realizados por desenvolvedores, entretanto é necessário que o sistema seja testado pelos clientes. No teste de aceitação, os clientes testam o sistema a fim de garantir

que o mesmo satisfaz suas necessidades. Vale destacar que o que foi especificado pelos desenvolvedores pode ser diferente do que queria o cliente. Assim, o teste de aceitação assegura que o sistema solicitado é o que foi construído.

- Teste de instalação: algumas vezes o teste de aceitação é feito no ambiente real de funcionamento, outras não. Quando o teste de aceitação for feito em um ambiente de teste diferente do local em que será instalado, é necessário realizar testes de instalação.

Referências

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.

PRESSMAN, R.S., *Engenharia de Software*, McGraw-Hill, 6ª edição, 2006.

MALDONADO, J. C., FABBRI, S.C.P.F., “Teste de Software”. In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.

Capítulo 8 – Entrega e Manutenção

Concluídos os testes, sistema aceito e instalado, estamos chegando ao fim do processo de desenvolvimento de software. A entrega é a última etapa desse processo. Uma vez entregue, o sistema passa a estar em operação e eventuais mudanças, sejam de caráter corretivo, sejam de caráter de evolução, caracterizam-se como uma manutenção.

8.1 - Entrega

A entrega não é meramente uma formalidade. No momento em que o sistema é instalado no local de operação e devidamente aceito, é necessário, ainda, ajudar os usuários a entenderem e a se sentirem mais familiarizados com o sistema. Neste momento, duas questões são cruciais para uma transferência bem-sucedida: treinamento e documentação (PFLEEGER, 2004).

A operação do sistema é extremamente dependente de pessoal com conhecimento e qualificação. Portanto, é essencial que o treinamento de pessoal seja realizado para que os usuários e operadores possam operar o sistema adequadamente.

A documentação que acompanha o sistema também tem papel crucial na entrega, afinal ela será utilizada como material de referência para a solução de problemas ou como informações adicionais. Essa documentação inclui, dentre outros, manuais do usuário e do operador, guia geral do sistema, tutoriais, ajuda (*help*), preferencialmente on-line e guias de referência rápida (PFLEEGER, 2004).

8.2 - Manutenção

O desenvolvimento de um sistema termina quando o produto é entregue para o cliente e entra em operação. A partir daí, deve-se garantir que o sistema continuará a ser útil e atendendo às necessidades do usuário, o que pode demandar alterações no mesmo. Começa, então, a fase de manutenção (SANCHES, 2001).

Há muitas causas para a manutenção, dentre elas (SANCHES, 2001): falhas no processamento devido a erros no software, falhas de desempenho, alterações no ambiente de dados, alterações no ambiente de processamento, necessidade de modificações em funções existentes e necessidade de inclusão de novas capacidades.

Ao contrário do que podemos pensar, a manutenção não é uma atividade trivial nem de pouca relevância. Ela é uma atividade importantíssima e de intensa necessidade de conhecimento. O mantenedor precisa conhecer o sistema, o domínio de aplicação, os requisitos do sistema, a organização que utiliza o mesmo, práticas de engenharia de software passadas e atuais, a arquitetura do sistema, algoritmos usados etc.

O processo de manutenção é semelhante, mas não igual, ao processo de desenvolvimento e pode envolver atividades de levantamento de requisitos, análise, projeto, implementação e testes, agora no contexto de um software existente. Essa

semelhança pode ser maior ou menor, dependendo do tipo de manutenção a ser realizada.

Pfleeger (PFLEEGER, 2004) aponta os seguintes tipos de manutenção:

- **Manutenção corretiva**: trata de problemas decorrentes de defeitos. À medida que falhas ocorrem, elas são relatadas à equipe de manutenção, que se encarrega de encontrar o defeito que causou a falha e faz as correções (nos requisitos, análise, projeto ou implementação), conforme o necessário. Esse reparo inicial pode ser temporário, visando manter o sistema funcionando. Quando esse for o caso, mudanças mais complexas podem ser implementadas posteriormente.
- **Manutenção adaptativa**: às vezes, uma mudança no ambiente do sistema, incluindo hardware e software de apoio, pode implicar em uma necessidade de adaptação.
- **Manutenção perfectiva**: consiste em realizar mudanças para melhorar algum aspecto do sistema, mesmo quando nenhuma das mudanças for conseqüência de defeitos. Isso inclui a adição de novas capacidades bem como ampliações gerais.
- **Manutenção preventiva**: consiste em realizar mudanças a fim de prevenir falhas. Geralmente ocorre quando um mantenedor descobre um defeito que ainda não causou falha e decide corrigi-lo antes que ele gere uma falha.

Referências

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2ª edição, 2004.

SANCHES, R., “Processo de Manutenção”. In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.